Code Explanation Assistance

i have viva and practical exam
if i provide you my code then can you help me by explaining the code line by line,also what is use of the code and how it works
and if possible can you give video to understand the concept in code

I can definitely help you with explaining the code line by line, detailing its functionality, and clarifying how it works. If you provide your code, I'll break it down and explain each part, along with the concepts behind it.

However, I can't create videos, but I can suggest how to better understand the concepts involved in the code through resources like tutorials or articles. If you're interested, I can help you find video resources as well.

Feel free to share your code, and we can get started!

```
#include<iostream>
using namespace std;

class node
{
public:
    string key;
    string meaning;
    node *left;
    node *right;
};

class AVL
{
    node *root;
       public:
               AVL()
               {
                   root=NULL;
               }

                void create();
                node* insert(node *cur,node *temp);
               node* balance(node *temp);
               int dif(node *temp);
               int height(node *temp);
```

```
        int maximum(int a,int b);

        node* LL(node *par);
        node* RR(node *par);
        node* LR(node *par);
        node* RL(node *par);

        void ascending(node *temp);
        node* delete_n(node *root,string key1);
        void deleten();

        node* extractmin(node *t);
        void descending(node *temp);
        void display();
        bool search(node *cur,string key1);
        void search_value();
};

void AVL::create()
{
  char answer;
  node *temp;
  do
  {
    temp=new node();
    cout<<endl<<"Enter keyword:\t";
    cin>>temp->key;
    cout<<"Enter meaning:\t";
    cin>>temp->meaning;
    temp->left=temp->right=NULL;

      root=insert(root,temp);

    cout<<endl<<"Add another word? (y/n):\t";
    cin>>answer;
  }
  while(answer=='y'||answer=='Y');
}


node* AVL::insert(node *cur,node *temp)
{
  if(cur==NULL)
  {
    return temp;
  }
  if(temp->key<cur->key)
  {
    cur->left=insert(cur->left,temp);
    cur=balance(cur);
  }
```

```
    else if(temp->key>cur->key)
    {
        cur->right=insert(cur->right,temp);
        cur=balance(cur);
    }
    return cur;
}

node* AVL::balance(node *temp)
{
    int bal;
    bal=dif(temp);

    if(bal>=2)
    {
        if(dif(temp->left)<0)
            temp=LR(temp);
        else
            temp=LL(temp);
    }
    else if(bal<=-2)
    {
        if(dif(temp->right)<0)
            temp=RR(temp);
        else
            temp=RL(temp);
    }
    return temp;
}


int AVL::dif(node *temp)
{
    int l,r;
    l=height(temp->left);
    r=height(temp->right);
    return(l-r);
}

int AVL::height(node *temp)
{
    if(temp==NULL)
        return(-1);
    else
        return(max(height(temp->left),height(temp->right))+1);
}

int AVL::maximum(int a,int b)
{
    if(a>b)
        return a;
```

```
    else
        return b;
}

node* AVL::LL(node *par)
{
    node *temp,*temp1;
    temp=par->left;
    temp1=temp->right;
    temp->right=par;
    par->left=temp1;
    return temp;
}

node* AVL::RR(node *par)
{
    node *temp,*temp1;
    temp=par->right;
    temp1=temp->left;
    temp->left=par;
    par->right=temp1;
    return temp;
}

node* AVL::LR(node *par)
{
    par->left=RR(par->left);
    return(LL(par));
}

node* AVL::RL(node *par)
{
    par->right=LL(par->right);
    return(RR(par));
}

void AVL::ascending(node *temp)
{
       if(temp!=NULL)
       {
               ascending(temp->left);
               cout<<"\n\t"<<temp->key<<" : "<<temp->meaning;
               ascending(temp->right);
       }
}

void AVL::descending(node *temp)
{
       if(temp!=NULL)
       {
               descending(temp->right);
```

```
            cout<<"\n\t"<<temp->key<<" : "<<temp->meaning;
            descending(temp->left);
        }
}


void AVL::display()
{
        cout<<endl<<"Keywords in ascending order:\t";
        ascending(root);
        cout<<endl<<"Keywords in descending order:\t";
        descending(root);
}

bool AVL::search(node *cur,string key1)
{
    if(cur)
    {
        if(cur->key==key1)
            return true;
        if(cur->key>key1)
            return search(cur->left,key1);
        else
            return search(cur->right,key1);
    }
    return false;
}

void AVL::search_value()
{
    string key2;
        cout<<endl<<"Keyword to search:\t";
        cin>>key2;
        if(search(root,key2))
                cout<<endl<<"Keyword exists in AVL tree.";
        else
                cout<<endl<<"Keyword does not exist in AVL tree.";
}


node* AVL::delete_n(node* cur,string key1)
{
   if ( !cur)
       return cur;
   if ( key1 < cur->key )
       cur->left = delete_n(cur->left, key1);

   else if( key1 > cur->key )
       cur->right = delete_n(cur->right, key1);

   else
```

```
    {
        node *l = cur->left;
        node *r = cur->right;
        delete cur;
        if ( !r )
           return l;
        node *m=r;

        while(m->left)
           m=m->left;
        m->right = extractmin(r);
        m->left = l;
        return balance(m);
    }
    return balance(cur);
}

    node* AVL::extractmin(node *t)
    {
        if ( !t->left )
        return t->right;
        t->left = extractmin(t->left);
        return balance(t);
    }

void AVL::deleten()
{
    string key;
    cout<<endl<<"Keyword to delete:\t";
    cin>>key;
    root=delete_n(root,key);
}

int main()
{
 char c;
 int ch;
 AVL a;
 do
 {
     cout<<endl<<"--- MAIN MENU ---";
     cout<<endl<<"1 -> Insert keyword";
     cout<<endl<<"2 -> Display AVL tree";
     cout<<endl<<"3 -> Search a keyword";
     cout<<endl<<"4 -> Delete a keyword";
     cout<<endl<<"Choose an option (1-4):\t";
     cin>>ch;
     switch(ch)
     {
         case 1 : a.create();
            break;
```

```
        case 2 : a.display();
          break;
        case 3 : a.search_value();
          break;
        case 4 : a.deleten();
          break;
        default : cout<<endl<<"Please choose a valid option (1-4).";
     }
     cout<<endl<<"Would you like to continue? (y/n):\t";
     cin>>c;
      }
      while(c=='y'||c=='Y');
     cout<<"\n\n// END OF CODE\n\n";
 return 0;
}
```

This code implements an **AVL tree** in C++, which is a self-balancing binary search tree. Here's a detailed explanation of the code:

Class Definitions:

- `node`: This class represents a single node in the AVL tree.
  - `key`: Stores the keyword (string type).
  - `meaning`: Stores the meaning of the keyword (string type).
  - `left` and `right`: Pointers to left and right child nodes in the tree.
- `AVL`: This class implements the AVL tree.
  - `root`: Pointer to the root node of the tree.
  - Several functions are defined within this class for inserting, balancing, deleting, displaying, and searching for words in the tree.

Constructor:

- `AVL()`: Initializes the `root` pointer to `NULL`, indicating the tree is initially empty.

Member Functions:

1. `create()`: This function allows the user to input keywords and their meanings and inserts them into the AVL tree. It runs in a loop, asking the user if they want to add another keyword.
2. `insert(node *cur, node *temp)`: This function recursively inserts a new node (`temp`) into the AVL tree starting at the node `cur`.
   - It compares the keyword of `temp` with `cur->key` and determines whether to go to the left or right subtree.
   - After inserting, it checks if the tree is balanced and performs necessary rotations.
3. `balance(node *temp)`: This function checks if the tree is balanced.
   - The difference between the heights of the left and right subtrees is calculated.

- If the difference is greater than or equal to 2 or less than or equal to -2, it performs appropriate rotations to balance the tree.

4. `` `dif(node *temp)` ``: Calculates the balance factor of the node `` `temp` `` by subtracting the height of the right subtree from the left subtree.
5. `` `height(node *temp)` ``: Returns the height of the subtree rooted at `` `temp` ``. The height is calculated as the maximum of the left and right subtree heights plus 1.
6. `` `maximum(int a, int b)` ``: A helper function that returns the maximum of two integers.
7. **Rotation Functions**: These are used to perform rotations to balance the AVL tree.
   - `` `LL(node *par)` ``: Left-Left rotation.
   - `` `RR(node *par)` ``: Right-Right rotation.
   - `` `LR(node *par)` ``: Left-Right rotation.
   - `` `RL(node *par)` ``: Right-Left rotation.
8. `` `ascending(node *temp)` ``: Performs an **in-order traversal** of the tree (left subtree -> current node -> right subtree) and displays the keywords and meanings in ascending order.
9. `` `descending(node *temp)` ``: Performs a **reverse in-order traversal** (right subtree -> current node -> left subtree) and displays the keywords and meanings in descending order.
10. `` `display()` ``: Calls the `` `ascending()` `` and `` `descending()` `` functions to display the tree's contents in ascending and descending order.
11. `` `search(node *cur, string key1)` ``: This function searches for a keyword (`` `key1` ``) in the tree starting from the node `` `cur` ``.
    - If the keyword matches `` `cur->key` ``, it returns `` `true` ``.
    - If `` `key1` `` is smaller, it searches the left subtree; if larger, it searches the right subtree.
12. `` `search_value()` ``: Prompts the user for a keyword to search and calls `` `search()` `` to find it in the AVL tree.
13. `` `delete_n(node* cur, string key1)` ``: This function deletes a node with the given keyword (`` `key1` ``).
    - It recursively searches for the node to delete.
    - If the node has two children, it replaces it with the minimum node in the right subtree.
    - It balances the tree after deletion.
14. `` `extractmin(node *t)` ``: This function extracts the minimum node from the subtree rooted at `` `t` `` (used for deletion).
15. `` `deleten()` ``: Prompts the user for a keyword to delete and calls `` `delete_n()` `` to delete the node from the tree.

Main Menu:

In the `` `main()` `` function:

- The user is presented with a menu to:
  - Insert a keyword and its meaning.
  - Display the AVL tree in ascending and descending order.

    - Search for a keyword.
    - Delete a keyword.
- The program continues running in a loop until the user decides to exit.

Concept of AVL Tree:

An **AVL tree** is a self-balancing binary search tree where the difference in heights of left and right subtrees of any node is at most 1. If, during an insertion or deletion, the tree becomes unbalanced, rotations are performed to restore balance.

- **Balance factor**: Difference between the heights of left and right subtrees.
    - ``balance factor = height(left subtree) - height(right subtree)``
    - A balance factor of 2 or -2 indicates that the tree is unbalanced and requires rotations.

Types of Rotations:

- **LL (Left-Left)**: The left subtree of the left child is too tall. A single right rotation is performed.
- **RR (Right-Right)**: The right subtree of the right child is too tall. A single left rotation is performed.
- **LR (Left-Right)**: The right subtree of the left child is too tall. A left rotation is followed by a right rotation.
- **RL (Right-Left)**: The left subtree of the right child is too tall. A right rotation is followed by a left rotation.

Example Walkthrough:

1. **Insertion**: When you insert a keyword (e.g., "apple"), the tree checks the left or right subtree based on lexicographical order and inserts the node.
2. **Balancing**: After inserting, the tree checks if it's unbalanced (e.g., if the balance factor is greater than 1 or less than -1), and performs rotations to keep the tree balanced.
3. **Display**: The ``ascending()`` function prints the keywords in lexicographical order, and the ``descending()`` function prints them in reverse order.

Let me know if you'd like further details on any part of the code or if you need help with understanding AVL tree concepts in more depth!