EXPERIMENT 01:

```
def display(s):
    print("{" + ", ".join(map(str, s)) + "}")

class SetOperations:
    def __init__(self):
        self.a = [int(input("Enter element of Set A: ")) for _ in range(int(input("Size of Set A: ")))]
        self.b = [int(input("Enter element of Set B: ")) for _ in range(int(input("Size of Set B: ")))]
        print("Set A:", self.a)
        print("Set B:", self.b)

    def union(self):
        display(list(set(self.a + self.b)))

    def intersection(self):
        display([x for x in self.a if x in self.b])

    def difference(self):
        while True:
            print("\nDifference\n1. A - B\n2. B - A\n3. Back")
            choice = int(input("Enter your choice: "))
            if choice == 1:
                display([x for x in self.a if x not in self.b])
            elif choice == 2:
                display([x for x in self.b if x not in self.a])
            elif choice == 3:
                break
            else:
                print("Invalid choice.")

    def subset(self):
        display([x for x in self.a if x in self.b])

# Main Program
s = SetOperations()
while True:
    print("\nMenu\n1. Union\n2. Intersection\n3. Difference\n4. Subset\n5. Exit")
    choice = int(input("Enter your choice: "))
    if choice == 1:
        s.union()
    elif choice == 2:
        s.intersection()
    elif choice == 3:
        s.difference()
    elif choice == 4:
        s.subset()
    elif choice == 5:
```

```
            break
        else:
            print("Invalid choice.")
```

EXPERIMENT 02:

```
class HashTable:
    def __init__(self):
        self.size = int(input("Enter number of phonebook users: "))
        self.table = [None] * self.size
        self.count = 0
        self.comparisons = 0

    def is_full(self):
        return self.count == self.size

    def insert(self):
        for _ in range(self.size):
            if self.is_full():
                print("Table is full.")
                break

            key = int(input("Enter key to insert: "))
            idx = key % self.size
            steps = 0

            # linear probing
            while self.table[idx] is not None and steps < self.size:
                self.comparisons += 1
                idx = (idx + 1) % self.size
                steps += 1

            if self.table[idx] is None:
                self.table[idx] = key
                self.count += 1
                self.comparisons += 1
                print(f"Inserted {key} at position {idx}")
            else:
                print("Could not insert: table seems full.")

    def display(self):
        print("\nHash Table:")
        for i, key in enumerate(self.table):
            print(f"{i}: {key}")

    def search(self):
        key = int(input("Enter key to search: "))
        idx = key % self.size
```

```
        steps = 0

        # probe until found or wrapped around
        while steps < self.size:
            self.comparisons += 1
            if self.table[idx] == key:
                print(f"Found {key} at position {idx}")
                return
            idx = (idx + 1) % self.size
            steps += 1

        print(f"{key} not found.")

if __name__ == "__main__":
    ht = HashTable()
    menu = {
        "1": ht.insert,
        "2": ht.display,
        "3": ht.search,
    }

    while True:
        print("\nMenu:\n1. Insert\n2. Display\n3. Search\n0. Exit")
        choice = input("Enter your choice: ")
        if choice == "0":
            break
        action = menu.get(choice)
        if action:
            action()
        else:
            print("Invalid choice.")
```

EXPERIMENT 03:

```
#include <iostream>
#include <string>
#include <vector>
#include <memory>

struct Node {
    std::string label;
    std::vector<std::unique_ptr<Node>> children;
};

class BookTree {
    std::unique_ptr<Node> root;

    // Recursive display with indentation
```

```
    void display(const Node* node, int indent = 0) const {
        if (!node) return;
        std::cout << std::string(indent, ' ') << node->label << "\n";
        for (const auto& child : node->children)
            display(child.get(), indent + 4);
    }

public:
    void create() {
        root = std::make_unique<Node>();
        std::cout << "Enter book title: ";
        std::getline(std::cin >> std::ws, root->label);

        int numChapters;
        std::cout << "Number of chapters: ";
        std::cin >> numChapters;

        for (int i = 1; i <= numChapters; ++i) {
            auto chap = std::make_unique<Node>();
            std::cout << " Chapter " << i << " title: ";
            std::getline(std::cin >> std::ws, chap->label);

            int numSections;
            std::cout << "  Number of sections in \"" << chap->label << "\": ";
            std::cin >> numSections;

            for (int j = 1; j <= numSections; ++j) {
                auto sec = std::make_unique<Node>();
                std::cout << "    Section " << j << " title: ";
                std::getline(std::cin >> std::ws, sec->label);
                chap->children.push_back(std::move(sec));
            }
            root->children.push_back(std::move(chap));
        }
    }

    void display() const {
        if (!root) {
            std::cout << "Tree is empty. Please create it first.\n";
        } else {
            std::cout << "\n--- Book Contents ---\n";
            display(root.get());
            std::cout << "----------------------\n";
        }
    }
};

int main() {
```

```
    BookTree tree;
    while (true) {
        std::cout << "\n1. Create  2. Display  3. Quit\nChoose: ";
        int choice; std::cin >> choice;
        switch (choice) {
            case 1: tree.create(); break;
            case 2: tree.display(); break;
            case 3: return 0;
            default: std::cout << "Invalid choice.\n";
        }
    }
}
```

EXPERIMENT 04:

```
#include <iostream>
#include <memory>

struct Node {
    int data;
    std::unique_ptr<Node> left, right;
    explicit Node(int v) : data(v) {}
};

class BST {
    std::unique_ptr<Node> root;

    // Recursive helpers
    std::unique_ptr<Node> insert(std::unique_ptr<Node> node, int v) {
        if (!node)
            return std::make_unique<Node>(v);
        if (v < node->data)
            node->left  = insert(std::move(node->left),  v);
        else
            node->right = insert(std::move(node->right), v);
        return node;
    }

    void inOrder(const Node* node) const {
        if (!node) return;
        inOrder(node->left.get());
        std::cout << node->data << ' ';
        inOrder(node->right.get());
    }

    int height(const Node* node) const {
        if (!node) return 0;
        return 1 + std::max(height(node->left.get()),
```

```
                    height(node->right.get()));
    }

    int minValue(const Node* node) const {
        return node->left ? minValue(node->left.get()) : node->data;
    }

    bool search(const Node* node, int v) const {
        if (!node) return false;
        if (node->data == v) return true;
        return search((v < node->data ? node->left.get() : node->right.get()), v);
    }

    void mirror(Node* node) {
        if (!node) return;
        std::swap(node->left, node->right);
        mirror(node->left.get());
        mirror(node->right.get());
    }

public:
    void insert(int v)        { root = insert(std::move(root), v); }
    void display() const      { inOrder(root.get()); std::cout << "\n"; }
    void printHeight() const  { std::cout << "Height: " << height(root.get()) << "\n"; }
    void printMin() const     {
        if (root) std::cout << "Min value: " << minValue(root.get()) << "\n";
        else      std::cout << "Tree is empty.\n";
    }
    void printSearch(int v) const {
        std::cout << (search(root.get(), v) ? "Found\n" : "Not found\n");
    }
    void makeMirror()         { mirror(root.get()); }
};

int main() {
    BST tree;
    int choice, value;

    do {
        std::cout << "\n1. Insert\n2. Display (In-order)\n"
                  "3. Height\n4. Min value\n"
                  "5. Mirror\n6. Search\n0. Exit\nChoose: ";
        std::cin >> choice;

        switch (choice) {
            case 1:
                std::cout << "Value to insert: ";
                std::cin >> value;
```

```
            tree.insert(value);
            break;
        case 2:
            tree.display();
            break;
        case 3:
            tree.printHeight();
            break;
        case 4:
            tree.printMin();
            break;
        case 5:
            tree.makeMirror();
            std::cout << "Tree mirrored.\n";
            break;
        case 6:
            std::cout << "Value to search: ";
            std::cin >> value;
            tree.printSearch(value);
            break;
        case 0:
            std::cout << "Goodbye!\n";
            break;
        default:
            std::cout << "Invalid choice.\n";
    }
  } while (choice != 0);

  return 0;
}
```

EXPERIMENT 05:

```
#include <iostream>
#include <string.h>
using namespace std;

struct node
{
  char data;
  node *left;
  node *right;
};
class tree
{
  char prefix[20];
```

```
public:
    node *top;
    void expression(char[]);
    void display(node *);
    void non_rec_postorder(node *);
    void del(node *);
};
class stack1
{
    node *data[30];
    int top;

public:
    stack1()
    {
        top = -1;
    }
    int empty()
    {
        if (top == -1)
            return 1;
        return 0;
    }
    void push(node *p)
    {
        data[++top] = p;
    }
    node *pop()
    {
        return (data[top--]);
    }
};
void tree::expression(char prefix[])
{
    char c;
    stack1 s;
    node *t1, *t2;
    int len, i;
    len = strlen(prefix);
    for (i = len - 1; i >= 0; i--)
    {
        top = new node;
        top->left = NULL;
        top->right = NULL;
        if (isalpha(prefix[i]))
        {
            top->data = prefix[i];
            s.push(top);
```

```
        }
        else if (prefix[i] == '+' || prefix[i] == '*' || prefix[i] == '-' || prefix[i] == '/')
        {
            t2 = s.pop();
            t1 = s.pop();
            top->data = prefix[i];
            top->left = t2;
            top->right = t1;
            s.push(top);
        }
    }
    top = s.pop();
}
void tree::display(node *root)
{
    if (root != NULL)
    {
        cout << root->data;
        display(root->left);
        display(root->right);
    }
}
void tree::non_rec_postorder(node *top)
{
    stack1 s1, s2; /*stack s1 is being used for flag . A NULL data implies that the right subtree
has not been visited */
    node *T = top;
    cout << "\n";
    s1.push(T);
    while (!s1.empty())
    {
        T = s1.pop();
        s2.push(T);
        if (T->left != NULL)
            s1.push(T->left);
        if (T->right != NULL)
            s1.push(T->right);
    }
    while (!s2.empty())
    {
        top = s2.pop();
        cout << top->data;
    }
}
void tree::del(node *node)
{
    if (node == NULL)
        return;
```

```
    /* first delete both subtrees */
    del(node->left);
    del(node->right);
    /* then delete the node */
    cout <<endl<<"Deleting node : " << node->data<<endl;
    free(node);
}
int main()
{
    char expr[20];
    tree t;

    cout <<"Enter prefix Expression : ";
    cin >> expr;
    cout << expr;
    t.expression(expr);
    //t.display(t.top);
    //cout<<endl;
   t.non_rec_postorder(t.top);
    t.del(t.top);
    // t.display(t.top);
}
```

EXPERIMENT 06:

```
#include <iostream>
#include <stdlib.h>
using namespace std;
int cost[10][10], i, j, k, n, u,v;
int stk[10], top, visit1[10], visited1[10];
int main()
{
    int m;
    cout << "Enter number of vertices : ";
    cin >> n;
    cout << "Enter number of edges : ";
    cin >> m;

    cout << "\nEDGES :\n";
    for (k = 1; k <= m; k++)
    {
        cout<<"Enter U and V:";
        cin >> i >> j;
        cost[i][j] = 1;
        cost[j][i] = 1;
    }
```

```
    //display function
    cout << "The adjacency matrix of the graph is : " << endl;
    for (i = 1; i <= n; i++)
    {
        for (j = 1; j <= n; j++)
        {
            cout << " " << cost[i][j];
        }
        cout << endl;
    }


    cout <<endl<<"Enter initial vertex : ";
    cin >> v;
    cout << "The DFS of the Graph is\n";
    cout << v;
    visited1[v] = 1;
    k = 1;
    while (k < n)
    {
        for (j = n; j >= 1; j--)
            if (cost[v][j] != 0 && visited1[j] != 1 && visit1[j] != 1)
            {
                visit1[j] = 1;
                stk[top] = j;
                top++;
            }
        v = stk[--top];
        cout << v << " ";
        k++;
        visit1[v] = 0;
        visited1[v] = 1;
    }
    return 0;
}
```

EXPERIMENT 07:

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


  int main()
{
        int n, i, j, k, row, col, mincost=0, min;
        char op;
        cout<<"Enter no. of vertices: ";
```

```
cin>>n;
int cost[n][n];
int visit[n];
for(i=0; i<n; i++)
	visit[i] = 0;
for(i=0; i<n; i++)
	for(int j=0; j<n; j++)
		cost[i][j] = -1;

for(i=0; i<n; i++)
{
	for(j=i+1; j<n; j++)
	{
		cout<<"Do you want an edge between "<<i<<" and "<<j<<": ";
		//use 'i' & 'j' if your vertices start from 0
		cin>>op;
		if(op=='y' || op=='Y')
		{
			cout<<"Enter weight: ";
			cin>>cost[i][j];
			cost[j][i] = cost[i][j];
		}
	}
}
visit[0] = 1;
for(k=0; k<n-1; k++)
{
	min = 999;
	for(i=0; i<n; i++)
	{
		for(j=0; j<n; j++)
		{
			if(visit[i] == 1 && visit[j] == 0)
			{
				if(cost[i][j] != -1 && min>cost[i][j])
				{
					min = cost[i][j];
					row = i;
					col = j;
				}
			}
		}
	}
	mincost += min;
	visit[col] = 1;
	cost[row][col] = cost[col][row] = -1;
	cout<<row<<"->"<<col<<endl;
	//use 'row' & 'col' if your vertices start from 0
```

```
        }
        cout<<"\nMin. Cost: "<<mincost;
        return 0;
}
```

EXPERIMENT 08:

```
#include<iostream>
using namespace std;
#define SIZE 10

class OBST {
    int p[SIZE];
    int q[SIZE];
    int a[SIZE];
    int w[SIZE][SIZE];
    int c[SIZE][SIZE];
    int r[SIZE][SIZE];
    int n;

public:
    void get_data() {
        int i;
        cout << "\n Optimal Binary Search Tree \n";
        cout << "\n Enter the number of nodes";
        cin >> n;
        cout << "\n Enter the data as…\n";
        for (i = 1; i <= n; i++) {
            cout << "\n a[" << i << "]";
            cin >> a[i];
        }
        for (i = 1; i <= n; i++) {
            cout << "\n p[" << i << "]";
            cin >> p[i];
        }
        for (i = 0; i <= n; i++) {
            cout << "\n q[" << i << "]";
            cin >> q[i];
        }
    }

    int Min_Value(int i, int j) {
        int m, k;
        int minimum = 32000;
        for (m = r[i][j - 1]; m <= r[i + 1][j]; m++) {
            if ((c[i][m - 1] + c[m][j]) < minimum) {
                minimum = c[i][m - 1] + c[m][j];
                k = m;
```

```
        }
    }
    return k;
}

void build_OBST() {
    int i, j, k, l, m;
    for (i = 0; i < n; i++) {
        w[i][i] = q[i];
        r[i][i] = c[i][i] = 0;
        w[i][i + 1] = q[i] + q[i + 1] + p[i + 1];
        r[i][i + 1] = i + 1;
        c[i][i + 1] = q[i] + q[i + 1] + p[i + 1];
    }
    w[n][n] = q[n];
    r[n][n] = c[n][n] = 0;
    for (m = 2; m <= n; m++) {
        for (i = 0; i <= n - m; i++) {
            j = i + m;
            w[i][j] = w[i][j - 1] + p[j] + q[j];
            k = Min_Value(i, j);
            c[i][j] = w[i][j] + c[i][k - 1] + c[k][j];
            r[i][j] = k;
        }
    }
}

void build_tree() {
    int i, j, k;
    int queue[20], front = -1, rear = -1;
    cout << "The Optimal Binary Search Tree For the Given Node Is…\n";
    cout << "\n The Root of this OBST is ::" << r[0][n];
    cout << "\nThe Cost of this OBST is::" << c[0][n];
    cout << "\n\n\t NODE \t LEFT CHILD \t RIGHT CHILD ";
    cout << "\n";
    queue[++rear] = 0;
    queue[++rear] = n;
    while (front != rear) {
        i = queue[++front];
        j = queue[++front];
        k = r[i][j];
        cout << "\n\t" << k;
        if (r[i][k - 1] != 0) {
            cout << "\t\t" << r[i][k - 1];
            queue[++rear] = i;
            queue[++rear] = k - 1;
        } else {
            cout << "\t\t";
```

```
            }
            if (r[k][j] != 0) {
                cout << "\t" << r[k][j];
                queue[++rear] = k;
                queue[++rear] = j;
            } else {
                cout << "\t";
            }
        }
        cout << "\n";
    }
};

int main() {
    OBST obj;
    obj.get_data();
    obj.build_OBST();
    obj.build_tree();
    return 0;
}
```

EXPERIMENT 09:

```
#include<iostream>
#include<cstring>
#include<cstdlib>
#define MAX 50
#define SIZE 20
using namespace std;

struct AVLnode
{
    public:
    char cWord[SIZE],cMeaning[MAX];
    AVLnode *left,*right;
    int iB_fac,iHt;
};

class AVLtree
{

    public:
        AVLnode *root;
        AVLtree()
        {
            root=NULL;
        }
        int height(AVLnode*);
```

```
        int bf(AVLnode*);
        AVLnode* insert(AVLnode*,char[SIZE],char[MAX]);
        AVLnode* rotate_left(AVLnode*);
        AVLnode* rotate_right(AVLnode*);
        AVLnode* LL(AVLnode*);
        AVLnode* RR(AVLnode*);
        AVLnode* LR(AVLnode*);
        AVLnode* RL(AVLnode*);
        AVLnode* delet(AVLnode*,char x[SIZE]);
        void inorder(AVLnode*);
};

AVLnode *AVLtree::delet(AVLnode *curr,char x[SIZE])
{
    AVLnode *temp;
    if(curr==NULL)
        return(0);
    else
        if(strcmp(x,curr->cWord)>0)
        {
            curr->right=delet(curr->right,x);
            if(bf(curr)==2)
            if(bf(curr->left)>=0)
                curr=LL(curr);
            else
                curr=LR(curr);
        }
        else
        if(strcmp(x,curr->cWord)<0)
        {
            curr->left=delet(curr->left,x);
            if(bf(curr)==-2)
            if(bf(curr->right)<=0)
                curr=RR(curr);
            else
                curr=RL(curr);
        }
    else
    {
        if(curr->right!=NULL)
        {
            temp=curr->right;
            while(temp->left!=NULL)
            temp=temp->left;
            strcpy(curr->cWord,temp->cWord);
            curr->right=delet(curr->right,temp->cWord);
            if(bf(curr)==2)
            if(bf(curr->left)>=0)
```

```
            curr=LL(curr);
         else
            curr=LR(curr);
      }
      else
      return(curr->left);
   }
   curr->iHt=height(curr);
   return(curr);
}


AVLnode* AVLtree :: insert(AVLnode*root,char newword[SIZE],char newmeaning[MAX])
{
   if(root==NULL)
   {
      root=new AVLnode;
      root->left=root->right=NULL;
      strcpy(root->cWord,newword);
      strcpy(root->cMeaning,newmeaning);
   }

   else if(strcmp(root->cWord,newword)!=0)
   {
      if(strcmp(root->cWord,newword)>0)
      {
         root->left=insert(root->left,newword,newmeaning);
         if(bf(root)==2)
         {
            if (strcmp(root->left->cWord,newword)>0)
               root=LL(root);
            else
               root=LR(root);
         }
      }

      else if(strcmp(root->cWord,newword)<0)
      {
         root->right=insert(root->right,newword,newmeaning);
         if(bf(root)==-2)
         {
            if(strcmp(root->right->cWord,newword)>0)
               root=RR(root);
            else
               root=RL(root);
         }
      }
   }
```

```
    else
        cout<<"\nRedundant AVLnode";
    root->iHt=height(root);
    return root;
}

int AVLtree :: height(AVLnode* curr)
{
    int lh,rh;
    if(curr==NULL)
        return 0;
    if(curr->right==NULL && curr->left==NULL)
        return 0;
    else
    {
        lh=lh+height(curr->left);
        rh=rh+height(curr->right);
        if(lh>rh)
            return lh+1;
        return rh+1;
    }
}

int AVLtree :: bf(AVLnode* curr)
{
    int lh,rh;
    if(curr==NULL)
        return 0;
    else
    {
        if(curr->left==NULL)
            lh=0;
        else
            lh=1+curr->left->iHt;
        if(curr->right==NULL)
            rh=0;
        else
            rh=1+curr->right->iHt;
        return(lh-rh);
    }
}

AVLnode* AVLtree :: rotate_right(AVLnode* curr)
{
    AVLnode* temp;
    temp=curr->left;
    curr->left=temp->right;
    temp->left=curr;
```

```
    curr->iHt=height(curr);
    temp->iHt=height(temp);
    return temp;
}

AVLnode* AVLtree :: rotate_left(AVLnode* curr)
{
    AVLnode* temp;
    temp=curr->right;
    curr->right=temp->left;
    temp->left=curr;
    curr->iHt=height(curr);
    temp->iHt=height(temp);
    return temp;
}

AVLnode* AVLtree :: RR(AVLnode* curr)
{
    curr=rotate_left(curr);
    return curr;
}

AVLnode* AVLtree :: LL(AVLnode* curr)
{
    curr=rotate_right(curr);
    return curr;
}

AVLnode* AVLtree :: RL(AVLnode* curr)
{
    curr->right=rotate_right(curr->right);
    curr=rotate_left(curr);
    return curr;
}

AVLnode* AVLtree::LR(AVLnode* curr)
{
    curr->left=rotate_left(curr->left);
    curr=rotate_right(curr);
    return curr;
}

void AVLtree :: inorder(AVLnode* curr)
{
    if(curr!=NULL)
    {
        inorder(curr->left);
        cout<<"\n\t"<<curr->cWord<<"\t"<<curr->cMeaning;
```

```
        inorder(curr->right);
    }
}

int main()
{
    int iCh;
    AVLtree a;
    AVLnode *curr=NULL;
    char cWd[SIZE],cMean[MAX];
    cout<<"\n--------------------------------------";
    cout<<"\n\tAVL TREE IMPLEMENTATION";
    cout<<"\n--------------------------------------";
    do
    {   cout<<"\n--------------------------------";
        cout<<"\n\t\tMENU";
        cout<<"\n--------------------------------";
        cout<<"\n1.Insert\n2.Inorder\n3.Delete\n4.Exit";
        cout<<"\n--------------------------------";
        cout<<"\nEnter your choice :";
        cin>>iCh;

        switch(iCh)
        {
            case 1: cout<<"\nEnter Word : ";
                cin>>cWd;
                cout<<"\nEnter Meaning : ";
                cin.ignore();
                cin.getline(cMean,MAX);

                a.root=a.insert(a.root,cWd,cMean);
                break;

            case 2: cout<<"\n\tWORD\tMEANING";
                a.inorder(a.root);
                break;

            case 3: cout<<"\nEnter the word to be deleted : ";
                    cin>>cWd;
                    curr=a.delet(a.root,cWd);
                    if(curr==NULL)
                        cout<<"\nWord not present!";
                    else
                        cout<<"\nWord deleted Successfully!";
                    curr=NULL;
                    break;

            case 4: exit(0);
```

```
        }
    }while(iCh!=4);

    return 0;
}
```

EXPERIMENT 10: