ASSIGNMENT 1

**GLOBAL VARIABLES**

size = 10

client\_list = [None] \* size

* size: The total size of the hash table (i.e., fixed number of slots for storing clients).
* client\_list: An array of 10 elements, initially all None, used to store client data.

**Function: add\_client()**

def add\_client():

* Defines a function to **add a client’s data** using **hashing with linear probing**.

client\_id = int(input("client id : "))

name = input("client name : ")

telephone = input("client telephone Number : ")

client\_details = [client\_id, name, telephone]

* Takes input for client ID, name, and telephone.
* Stores them in a list called client\_details.

index = client\_id % size

* Calculates the hash index using **modulo operation**.

for i in range(size):

if client\_list[index] == None:

client\_list[index] = client\_details

print("Client details added as : ", index, client\_details)

break

else:

index = (index + 1) % size

* Uses **linear probing** to resolve collisions:
  + If the calculated index is occupied, move to the next one.
  + Wraps around using modulo (% size).
  + Adds data to the first available slot.

print("\n Client List :",client\_list)

* Displays the full client list after insertion.

**Function: search\_client()**

def search\_client():

* Function to search for a client based on client ID.

client\_id = int(input("Enter client id to be search : "))

index = client\_id % size

cnt=0

* Takes input, calculates hash index, and initializes comparison counter cnt.

for i in range(size):

if client\_list[index] != None:

cnt=cnt+1

if client\_list[index][0] == client\_id:

print("client is a found at index ", index, client\_list[index])

print("\n Number of comparisions required to search client id %d are %d."%(client\_id,cnt))

break

index = (index + 1) % size

* Linearly probes through the array to find the client.
* Stops when the client is found or the table is fully checked.
* Counts and displays the number of comparisons needed.

else:

print("element is not found")

print("\n Number of comparisions required to search client id %d are %d."%(client\_id,cnt))

* If not found, displays a message and prints the number of comparisons.

**Function: delete\_client()**

def delete\_client():

client\_id = int(input("client id"))

index = client\_id % size

* Takes client ID input and calculates its index.

for i in range(size):

if client\_list[index] != None:

if client\_list[index][0] == client\_id:

client\_list[index] = None

print("cliet delted")

break

index = (index + 1) % size

* If match is found during probing, sets the slot to None (deletes).
* **Note**: This can create search issues later, as None might incorrectly signal “never inserted” instead of “was deleted”.

else:

print("element is not found")

* If client is not found in entire loop, prints not found.

**Function: Main()**

def Main() :

Group\_A = []

Group\_B = []

Group\_C = []

* Defines the main function and initializes 3 unused groups (likely leftover or future use).

while True :

print ("\t1 : ADD Client")

print ("\t2 : Serach Client")

print ("\t3: Exit")

ch = int(input("Enter your choice : "))

Group\_R = []

* Loops menu with options for adding, searching, or exiting.
* Group\_R is defined but unused.

if (ch == 3):

print ("End of Program")

break

* Exits loop if choice is 3.

elif (ch==1):

add\_client()

elif (ch==2):

search\_client()

* Calls add\_client() or search\_client() based on choice.

else :

print ("Wrong choice entered !! Try again")+ # <- this line has an error

* Syntax Error: + at the end is incorrect.

**Algorithm Summary: Hashing with Linear Probing**

* **Hash Function**: client\_id % size
* **Insertion**:
  + If index is empty → insert
  + Else → probe next slots linearly (circular with % size)
* **Search**:
  + Start from hash index, linearly search until match or all slots checked.
* **Delete**:
  + Same linear search logic as search, then set the slot to None if found.

ASSIGNMENT 2

Sure! Here is a **detailed line-by-line explanation** of the second program, including **the meaning of each line and what it does**, written in a simple and clear way.

**✅ Globals**

size = 10

* Defines the size of the hash table as 10.
* This is the maximum number of clients that can be stored.

**🔹 Function: insert\_data(client\_list)**

def insert\_data(client\_list):

* This function inserts a new client into the hash table.

il = int(input("Enter the id of client:"))

* Takes the client's ID as integer input from the user.

name = input("Enter the name of client:")

* Takes the client's name as input.

telephone = input("Enter the Telephone number of client:")

* Takes the client's telephone number.

x = il % 10

* Computes the hash index using the modulo operator (id % size).
* This is the **home address** where the client should be placed initially.

chain = None

* Initializes the chain (next link) as None. This will later be updated if needed.

p = x

* Stores the original hash index (home address) in variable p for future use.

while(client\_list[x] != None):

x = x + 1

* If the spot at index x is occupied (collision), move to the next index (linear probing) until a free slot is found.

client\_list[x] = [il, name, telephone, chain]

* Stores the client data at the calculated index x as a list:
  + [ID, name, telephone, link\_to\_next\_same\_hash].

m = client\_list[p]

* Stores the current data at the home index p into m (to use in case we need to swap it later).

**🔁 Replacement Handling**

if (p < x):

if (client\_list[p][0] % 10 != client\_list[x][0] % 10):

* Checks if the element at the home index p **does not belong** to that position.
* In that case, **swap the new correct client in**, and move the old one to a new position.

client\_list[p] = [il, name, telephone, chain]

client\_list[x] = m

* Swaps the two clients as described above.

y = client\_list[x][0] % size

* Gets the hash index of the displaced client (now at x) to find where its chain begins.

while(client\_list[y][3] == client\_list[x][0]):

y = client\_list[y][3]

* Walks through the chain of the displaced client to reach the last link.

client\_list[y][3] = x

* Adds x as the next link in the chain.

**🔗 Normal Chaining (No Replacement)**

if (p < x):

if (client\_list[p][0] % 10 == client\_list[x][0] % 10):

* If the inserted client and the one at home index both **belong** to the same home bucket, then:

while(client\_list[p][3] != None):

p = client\_list[p][3]

* Walk through the chain to find the last node.

client\_list[p][3] = x

* Add new client to the end of the chain.

**🔹 Function: search(client\_list, key)**

def search(client\_list, key):

* Searches for a client using their ID (key).

x = key % 10

* Calculates the starting index (home address) from the ID.

while(client\_list[x] != None and key != client\_list[x][0] and client\_list[x][3] != None):

x = client\_list[x][3]

* Walks through the chain of indexes if:
  + Current slot is not empty,
  + The key doesn’t match current client ID,
  + And there is a next link to follow.

if(client\_list[x] != None and key == client\_list[x][0]):

print("client found at", x, " index")

* If the current slot has the client with the matching key, it is found.

else:

print(key, " Not found....")

* If client not found after the chain ends.

**🔹 Function: delete(client\_list)**

def delete(client\_list):

* Deletes a client from the table.

key = int(input("Enter the id of deleted client:"))

* Reads the ID of the client to be deleted.

x = key % 10

p = x

* Calculates the home index and stores it in x and p.

while(key != client\_list[x][0]):

x = client\_list[x][3]

* Walks through the chain until the client with the given ID is found.

while(client\_list[p][3] != x):

p = client\_list[p][3]

* Finds the previous node (p) that points to the current node (x) for unlinking.

client\_list[p][3] = None

* Removes the current node from the chain.

if(client\_list[x][3] != None):

client\_list[p][3] = client\_list[x][3]

* If the deleted node had a child, links it to the previous node.

client\_list[x] = None

* Clears the client data from the hash table.

**🔹 Function: main()**

def main():

client\_list = [None] \* size

* Initializes an empty list of size 10 to store client data.

while True:

* Infinite loop to keep showing the menu until the user exits.

print("1.Insert the data:")

print("2.Display the data:")

print("3.Search the data:")

print("4.Delete the data:")

print("5.Exit")

ch = int(input("Enter your choise:"))

* Displays menu and reads user input.

if(ch == 1):

insert\_data(client\_list)

* Calls the function to insert a new client.

elif(ch == 2):

print("client list:\n")

print(client\_list)

* Displays the complete client list (raw format).

elif(ch == 3):

key = int(input("Enter the key for searching the client:"))

search(client\_list, key)

* Prompts for a client ID and searches for it.

elif(ch == 4):

delete(client\_list)

* Calls the function to delete a client.

elif(ch == 5):

break

* Exits the loop and ends the program.

else:

print("Invalid Input....")

* Error message for invalid choice.

main()

* Calls the main function to start the program.

**✅ Summary of Data Structure**

Each client\_list[x] holds a list of the form:

[client\_id, client\_name, telephone, next\_link\_index]

* next\_link\_index is used to chain elements with the **same hash key** using **index pointers**.

ASSIGNMENT 3

Here's a complete **line-by-line explanation** and **algorithm** for the C++ program you posted. This program uses a **Generalized Linked List (GLL)** structure to represent a book, its chapters, sections, and subsections in a hierarchical way.

**🔧 DATA STRUCTURE USED**

struct node {

char name[20]; // stores name of the book/chapter/section/subsection

node \*next; // pointer to next node at the same level

node \*down; // pointer to the first node in the next level

int flag; // 0 if no children, 1 if children exist

};

**📚 CLASS DEFINITION**

class Gll {

char ch[20];

int n, i;

node \*head = NULL, \*temp = NULL, \*t1 = NULL, \*t2 = NULL;

public:

node\* create();

void insertb();

void insertc();

void inserts();

void insertss();

void displayb();

};

This Gll class handles:

* insertb(): Insert book
* insertc(): Insert chapter(s)
* inserts(): Insert section(s)
* insertss(): Insert subsection(s)
* displayb(): Display the full hierarchy

**⚙️ Function: create()**

Creates a new node with user input.

node\* Gll::create() {

node\* p = new node;

p->next = NULL;

p->down = NULL;

p->flag = 0;

cout << "\nEnter the name: ";

cin >> p->name;

return p;

}

* Allocates memory
* Initializes pointers to NULL and flag to 0
* Takes input for the name

**📘 Function: insertb() — Insert Book**

void Gll::insertb() {

if (head == NULL) {

t1 = create();

head = t1;

} else {

cout << "\nBook already exists!";

}

}

* Only one book allowed (i.e., single root node).
* Creates a book node if not already created.

**📄 Function: insertc() — Insert Chapters**

void Gll::insertc() {

if (head == NULL) {

cout << "\nThere is no book.";

} else {

cout << "\nHow many chapters do you want to insert: ";

cin >> n;

for (i = 0; i < n; i++) {

t1 = create();

if (head->flag == 0) {

head->down = t1;

head->flag = 1;

} else {

temp = head->down;

while (temp->next != NULL)

temp = temp->next;

temp->next = t1;

}

}

}

}

* Inserts n chapters under the book.
* If it’s the first chapter, link it using down.
* Others are added using next.

**📑 Function: inserts() — Insert Sections under a Chapter**

void Gll::inserts() {

...

}

* Asks user for chapter name where sections are to be inserted.
* Searches chapter via down pointers.
* On finding the chapter, inserts n sections:
  + First section is added via down.
  + Others via next.

**📂 Function: insertss() — Insert Subsections under a Section**

void Gll::insertss() {

...

}

* Takes input: chapter name and section name
* Navigates to the section under that chapter
* Inserts subsections below the section using same pattern:
  + First: down
  + Rest: next

**📊 Function: displayb() — Display the Entire Structure**

void Gll::displayb() {

if (head == NULL) {

cout << "\nBook not exists.";

} else {

...

}

}

Displays the hierarchy:

* Book → Chapter(s) → Section(s) → Subsection(s)
* Uses nested loops and pointer traversal

**🧭 Main Menu:**

int main() {

...

}

Menu options:

1. Insert book
2. Insert chapters
3. Insert sections
4. Insert subsections
5. Display all
6. Exit

Each choice maps to the relevant class method.

**🧠 ALGORITHM SUMMARY**

1. create() - makes a node with name, NULL pointers, flag = 0

2. insertb() - if book doesn't exist, create it and set as head

3. insertc()

- Input number of chapters

- If head->flag == 0, set head->down to first chapter

- Else traverse using next and add new chapters at the end

4. inserts()

- Input chapter name

- Traverse to chapter via down/next

- Insert sections under found chapter (down or next)

5. insertss()

- Input chapter name → section name

- Traverse down and next to locate section

- Insert subsections under it

ASSIGNMENT 4

Here is the **line-by-line logic** and **algorithm** explanation for your complete C++ program that implements a **Binary Search Tree (BST)** with operations like insert, traversal, search, mirror, height, and minimum element.

**🧱 STRUCTURE DEFINITION**

struct node {

int data;

node \*L;

node \*R;

};

**🔍 Logic:**

* This is a **binary tree node** with three parts:
  + data: stores the value.
  + L: pointer to the left child.
  + R: pointer to the right child.

**🧠 CLASS DEFINITION: bst**

**CONSTRUCTOR**

bst() {

root = NULL;

count = 0;

}

* Initializes root to NULL (empty tree) and count to 0.

**FUNCTION: create()**

void bst::create() {

char ans;

do {

temp = new node; // allocate memory for a new node

cout << "Enter the data : ";

cin >> temp->data; // input node value

temp->L = NULL;

temp->R = NULL; // set left and right to NULL

if (root == NULL) // if tree is empty

root = temp; // set first node as root

else

insert(root, temp); // insert into tree using insert()

cout << "Do you want to insert more value (y/n)? ";

cin >> ans;

count++; // increase node count

cout << endl;

} while (ans == 'y');

cout << "The Total no. of nodes are: " << count << endl;

}

**🔄 Algorithm:**

1. Repeat until user stops:
   * Create a new node.
   * If tree is empty, set it as root.
   * Otherwise, call insert() to place it correctly.
   * Count each new node.

**FUNCTION: insert(root, temp)**

void bst::insert(node \*root, node\* temp) {

if (temp->data > root->data) { // if new data is greater

if (root->R == NULL) // right child empty

root->R = temp; // insert to right

else

insert(root->R, temp); // recurse on right

} else {

if (root->L == NULL) // if less or equal

root->L = temp; // insert to left

else

insert(root->L, temp); // recurse on left

}

}

**🌱 Algorithm:**

* Recursively find the correct position in BST:
  + Go **right** if value is greater.
  + Go **left** otherwise.

**FUNCTION: disin(node \*root) – Inorder Traversal**

void bst::disin(node \*root) {

if (root != NULL) {

disin(root->L);

cout << root->data << "\t";

disin(root->R);

}

}

**🔁 Algorithm (L → N → R):**

1. Recursively print left subtree.
2. Print current node.
3. Recursively print right subtree.

**FUNCTION: dispre() – Preorder Traversal**

void bst::dispre(node \*root) {

if (root != NULL) {

cout << root->data << "\t";

dispre(root->L);

dispre(root->R);

}

}

**🔁 Algorithm (N → L → R):**

1. Print current node.
2. Recursively print left subtree.
3. Recursively print right subtree.

**FUNCTION: dispost() – Postorder Traversal**

void bst::dispost(node \*root) {

if (root != NULL) {

dispost(root->L);

dispost(root->R);

cout << root->data << "\t";

}

}

**🔁 Algorithm (L → R → N):**

1. Recursively print left subtree.
2. Recursively print right subtree.
3. Print current node.

**FUNCTION: search()**

void bst::search(node \*root, int key) {

int flag = 0;

temp = root;

while (temp != NULL) {

if (key == temp->data) {

cout << " KEY FOUND \n";

flag = 1;

break;

}

if (key > temp->data)

temp = temp->R;

else

temp = temp->L;

}

if (flag == 0)

cout << " KEY NOT FOUND " << endl;

}

**🔍 Algorithm:**

* Start at root.
* Loop:
  + If current node matches key → print found.
  + If key > node → go right.
  + If key < node → go left.
  + Stop when node becomes NULL.
* If not found, print "not found".

**FUNCTION: height()**

int bst::height(node \*root) {

int hl, hr;

if (root == NULL)

return 0;

else if (root->L == NULL && root->R == NULL)

return 1;

hr = 1 + height(root->R);

hl = 1 + height(root->L);

return (hr > hl) ? hr : hl;

}

**📏 Algorithm:**

* Recursively find height of left and right subtrees.
* Return 1 + max(left, right).
* Height of empty tree is 0.
* Height of single node is 1.

**FUNCTION: min()**

void bst::min(node \*root) {

temp = root;

while (temp->L != NULL)

temp = temp->L;

cout << temp->data << endl;

}

**📉 Algorithm:**

* Go left until L == NULL → the smallest value in BST.

**FUNCTION: mirror()**

void bst::mirror(node \*root) {

temp = root;

if (root != NULL) {

mirror(root->L); // recurse on left

mirror(root->R); // recurse on right

temp = root->L; // swap left and right

root->L = root->R;

root->R = temp;

}

}

**🔁 Algorithm:**

* Recursively go to each node.
* Swap its left and right child.

**FUNCTION: getRoot()**

node\* bst::getRoot() {

return root;

}

**🔧 Purpose:**

* Used in main() to access the private root of the BST.

**💡 MAIN MENU LOGIC**

do {

cout << "\n1) Insert...\n2) Height...\n...";

cin >> ch;

switch(ch) {

case 1: t.create(); break;

case 2: t.height(t.getRoot()); break;

...

}

cout << "Do you want to continue (y/n)? ";

cin >> ans;

} while(ans == 'y');

**🔂 Repeats until user chooses to stop.**

* Switch handles each operation:
  + Create tree
  + Find height
  + Find min
  + Mirror tree
  + Search key
  + Traverse in inorder / preorder / postorder

ASSIGNMENT 5

Certainly! Here's the **algorithm** for each major operation in the dictionary program using a **Binary Search Tree (BST)**:

**🔹 1. Create Dictionary**

**Algorithm: CREATE()**

Step 1: Repeat until user wants to stop

a. Allocate memory for a new node (temp)

b. Input keyword (k) and meaning (m) from the user

c. Set temp->left = NULL, temp->right = NULL

d. If root is NULL

Set root = temp

Else

Call INSERT(root, temp)

Step 2: Ask user if they want to add more

**🔹 2. Insert Node in BST**

**Algorithm: INSERT(root, temp)**

Step 1: If temp->k < root->k

If root->left is NULL

root->left = temp

Else

Call INSERT(root->left, temp)

Step 2: Else

If root->right is NULL

root->right = temp

Else

Call INSERT(root->right, temp)

**🔹 3. Display Dictionary (Inorder Traversal)**

**Algorithm: DISP(root)**

Step 1: If root is not NULL

a. Call DISP(root->left)

b. Print root->k and root->m

c. Call DISP(root->right)

**🔹 4. Search for a Keyword**

**Algorithm: SEARCH(root, key)**

Step 1: Initialize comparison count c = 0

Step 2: While root is not NULL

a. Increment c

b. If key == root->k

Print comparisons

Return 1 (found)

c. Else if key < root->k

root = root->left

d. Else

root = root->right

Step 3: Return -1 (not found)

**🔹 5. Update Meaning of a Keyword**

**Algorithm: UPDATE(root, key)**

Step 1: While root is not NULL

a. If key == root->k

Input new meaning

root->m = new meaning

Return 1 (updated)

b. Else if key < root->k

root = root->left

c. Else

root = root->right

Step 2: Return -1 (not found)

**🔹 6. Delete a Keyword**

**Algorithm: DEL(root, key)**

Step 1: If root is NULL

Print "Element Not Found"

Return NULL

Step 2: If key < root->k

root->left = DEL(root->left, key)

Else if key > root->k

root->right = DEL(root->right, key)

Else (node to delete found)

a. If root has no children

Delete root

Return NULL

b. If root has only right child

Save root->right in temp

Delete root

Return temp

c. If root has only left child

Save root->left in temp

Delete root

Return temp

d. If root has two children

Find minimum node in right subtree (MIN(root->right))

Copy min node's keyword and meaning to root

root->right = DEL(root->right, min->k)

Step 3: Return root

**🔹 7. Find Minimum Node**

**Algorithm: MIN(root)**

Step 1: While root->left is not NULL

root = root->left

Step 2: Return root

ASSIGNMENT 6

Here is a **line-by-line logic explanation and algorithm** of your program along with time complexities, including a breakdown of the function displaya().

**🔹 Class Purpose**

The program creates a **graph representation** of cities using both:

* An **adjacency matrix** (m[n][n])
* An **adjacency list** (head[i] linked list for each vertex)

Each edge has a **weight (time in minutes)** representing the time to travel between cities.

**🧠 Function-by-Function Algorithm and Complexity**

**✅ 1. getgraph()**

**Purpose:** Accepts the number of cities and builds the adjacency matrix with travel times.

**Algorithm:**

Step 1: Input number of cities n

Step 2: Input city names v[0] to v[n-1]

Step 3: For i = 0 to n-1

For j = 0 to n-1

Ask if a path exists from v[i] to v[j]

If yes, input time and store in m[i][j]

If no or invalid, set m[i][j] = 0

Step 4: Call adjlist() to build the adjacency list

**Time Complexity:** O(n²)  
(Loop over all city pairs)

**✅ 2. adjlist()**

**Purpose:** Builds an adjacency list from the matrix.

**Algorithm:**

Step 1: For i = 0 to n-1

Create a head node for v[i]

Set head[i] = new node with vertex = v[i]

Step 2: For i = 0 to n-1

For j = 0 to n-1

If m[i][j] != 0

Create a new node for v[j]

Append it to the end of head[i]'s list

**Time Complexity:**

* Best/Average: O(n²) (matrix traversal)
* Worst Case: O(n³) if appending to linked list takes O(n) time (but in practice it's closer to O(n²) unless many connections per node)

**✅ 3. displaym()**

**Purpose:** Displays the adjacency matrix with city names.

**Algorithm:**

Step 1: Print column headers using v[]

Step 2: For i = 0 to n-1

Print row label (v[i])

For j = 0 to n-1

Print m[i][j]

**Time Complexity:** O(n²)

**✅ 4. displaya()**

**Purpose:** Displays the adjacency list and travel times.

**Algorithm:**

Step 1: For i = 0 to n-1

If head[i] is NULL

Print "No adjacency list"

Else

Print head[i]->vertex

Traverse through head[i]->next and print all vertex names

Step 2: For i = 0 to n-1

If head[i] is NULL

Print "No adjacency list"

Else

Traverse through head[i]->next

For each node, print:

head[i]->vertex -> temp->vertex [Time: temp->time]

**Time Complexity:**

* Step 1: Traverse each list → O(n + e)
* Step 2: Again traverses each edge → O(n + e)

✅ **Total Complexity:** O(n + e)  
where:

* n = number of cities (vertices)
* e = number of edges (connections between cities)

**🧾 Summary Table**

| **Function** | **Purpose** | **Time Complexity** |
| --- | --- | --- |
| getgraph() | Input cities and travel times | O(n²) |
| adjlist() | Convert matrix to list | O(n²) – O(n³)\* |
| displaym() | Display adjacency matrix | O(n²) |

ASSIGNMENT 7

Here's a **detailed algorithm and time complexity breakdown** for your program that implements **Prim's Algorithm** using an **adjacency matrix** to find the **Minimum Spanning Tree (MST)** for a given network of branches and connections.

**🔍 Purpose of the Program**

* Input a **weighted undirected graph** (branches and phone line charges).
* Store it as an **adjacency matrix**.
* Find the **Minimum Spanning Tree (MST)** using **Prim's algorithm**.
* Display the minimum total cost and selected connections.

**✅ FUNCTION-BY-FUNCTION EXPLANATION AND ALGORITHM**

**🔹 input()**

**Purpose:** Accepts the number of vertices and connections, and fills the adjacency matrix with weights.

**Algorithm:**

Step 1: Input number of vertices (branches) → v

Step 2: Initialize a[v][v] with 999 (used as INF)

Initialize visited[i] = 0 for all i

Step 3: Input number of edges (connections) → e

Step 4: For i = 0 to e-1:

Input l and u (connected vertices, 1-based)

Input weight (charge) w

Set a[l-1][u-1] = a[u-1][l-1] = w

**Time Complexity:** O(e)  
(Loop runs for all edges to set weights)

**🔹 display()**

**Purpose:** Prints the adjacency matrix representing the graph.

**Algorithm:**

Step 1: Loop i from 0 to v-1

Loop j from 0 to v-1

Print a[i][j]

**Time Complexity:** O(v²)  
(Full matrix is printed)

**🔹 minimum() → Prim’s Algorithm**

**Purpose:** Finds the **minimum total cost** to connect all vertices using **Prim's MST algorithm**.

**Algorithm:**

Step 1: Mark visited[1] = 1 (or any arbitrary starting point)

Step 2: For count = 0 to v-2:

min = INF

For i = 0 to v-1:

If visited[i] == 1:

For j = 0 to v-1:

If visited[j] == 0 and a[i][j] < min:

Update min = a[i][j], p = i, q = j

Mark visited[q] = 1

Add min to total

Output connection p+1 -> q+1 and its cost

Step 3: Print total cost

**Time Complexity:**  
✅ O(v²)  
Because:

* Outer loop runs (v - 1) times
* Each time, it scans the full adjacency matrix (nested loop)

**🧾 Summary Table**

| **Function** | **Purpose** | **Time Complexity** |
| --- | --- | --- |
| input() | Take graph input and fill matrix | O(e) |
| display() | Show adjacency matrix | O(v²) |
| minimum() | Prim’s algorithm (MST) | O(v²) |

✅ This version of Prim's algorithm is **suitable for dense graphs** (matrix form).  
🟡 For **sparse graphs**, a priority queue and adjacency list would reduce it to **O((v + e) log v)**.

**🔚 Final Notes**

* visited[1]=1; → You could also start with visited[0]=1 for simplicity, since arrays are 0-indexed.
* The use of 999 as **INF** is fine for small weights, but for clarity and safety, you might define const int INF = 1e9;

ASSIGNMENT 8

Here’s a **detailed explanation and time complexity analysis** for your program that implements the **Optimal Binary Search Tree (OBST)** using **dynamic programming**.

**🔍 Purpose of the Program**

The program constructs an **Optimal Binary Search Tree (OBST)** from given keys, their associated search probabilities (successful and unsuccessful), and then calculates the **minimum search cost** for the tree structure using dynamic programming.

**✅ FUNCTION-BY-FUNCTION EXPLANATION AND ALGORITHM**

**🔹 accept()**

**Purpose:** Accepts the input of keys, probabilities of successful searches (p[i]), and probabilities of unsuccessful searches (q[i]).

**Algorithm:**

Step 1: Input number of elements (keys) → n

Step 2: Input keys (a[i]) → For i = 1 to n

Step 3: Input probabilities of successful searches (p[i]) → For i = 1 to n

Step 4: Input probabilities of unsuccessful searches (q[i]) → For i = 0 to n

**Time Complexity:** O(n)  
(Looping to input the keys and probabilities)

**🔹 cons\_obst()**

**Purpose:** Constructs the Optimal Binary Search Tree using dynamic programming by minimizing the search cost.

**Algorithm:**

Step 1: Initialize w, c, and r arrays:

- Set w[i][i] = q[i]

- Set c[i][i] = r[i][i] = 0

- Set w[i][i+1] = q[i] + p[i+1] + q[i+1]

- Set c[i][i+1] = w[i][i+1], r[i][i+1] = i+1

- Set w[n][n] = q[n], c[n][n] = r[n][n] = 0

Step 2: Use dynamic programming to compute w, c, and r for larger subtrees:

For m = 2 to n (size of the subproblem):

For i = 0 to n-m:

j = i + m

w[i][j] = w[i][j-1] + p[j] + q[j]

k = find\_min(i, j) // Find the optimal root for the subtree

c[i][j] = w[i][j] + c[i][k-1] + c[k][j]

r[i][j] = k

Step 3: Output the root of the OBST (a[r[0][n]]), and recursively display the left and right subtrees:

Call tree(0, r[0][n]-1) for left subtree

Call tree(r[0][n], n) for right subtree

**Time Complexity:** O(n³)

* **Outer loop:** Runs for m from 2 to n, so O(n).
* **Inner loop:** Runs for i from 0 to n-m, and for each, performs find\_min(i, j).

**Key operation:** find\_min(i, j) inside the nested loop.

**🔹 find\_min(int i, int j)**

**Purpose:** Finds the optimal root index (k) that minimizes the cost of the subtree between i and j.

**Algorithm:**

Step 1: Initialize `min = 999999` and set `z = r[i][j-1]`

Step 2: For k = r[i][j-1] to r[i+1][j]:

- Compute cost = c[i][k-1] + c[k][j]

- If the cost is less than `min`, update `min` and `z`

Step 3: Return the index `z` of the optimal root

**Time Complexity:** O(n)

* The loop runs from r[i][j-1] to r[i+1][j] which is typically O(n) in the worst case.

**🔹 tree(int i, int j)**

**Purpose:** Recursively prints the structure of the tree, displaying the root, left subtree, and right subtree.

**Algorithm:**

Step 1: If r[i][j] == 0, print "NULL" (Base case)

Step 2: Otherwise, print the root node `a[r[i][j]]`

Step 3: Recursively print left and right subtrees:

- Left child: tree(i, r[i][j] - 1)

- Right child: tree(r[i][j], j)

**Time Complexity:** O(n)

* In the worst case, we print each key once, so the complexity is linear in terms of n.

**Main Function**

1. Calls the accept() function to get the input from the user.
2. Calls the cons\_obst() function to construct the optimal binary search tree and find the minimum search cost.
3. Displays the root and the subtrees using tree().

**🧾 Summary Table**

| **Function** | **Purpose** | **Time Complexity** |
| --- | --- | --- |
| accept() | Take inputs for keys and probabilities | O(n) |
| cons\_obst() | Construct the Optimal Binary Search Tree | O(n³) |
| find\_min(i, j) | Find optimal root index for subtree | O(n) |
| tree(i, j) | Recursively display the tree structure | O(n) |

**🔚 Final Notes**

* **Dynamic Programming:** The algorithm is efficient for **small to medium-sized inputs** but has **O(n³)** complexity, which could be slow for very large n.
* **Recursive Tree Printing:** The tree() function is recursive, and handles the structure of the tree by navigating through r[i][j].
* The solution can be optimized further for larger datasets by using **memoization** or more advanced tree-building techniques, but this is a standard dynamic programming solution.

ASSIGNMENT 9

Here's a **detailed explanation and time complexity analysis** of your AVL tree-based **Dictionary program**, which includes operations like insertion, deletion, searching, and modifying entries, as well as traversals such as preorder, inorder, and postorder.

**🔍 Program Overview**

The program uses an **AVL Tree** (a self-balancing binary search tree) to store words (keywords) and their meanings. The tree ensures that the height of the left and right subtrees of any node differ by at most 1, making search operations efficient. The program supports the following functionalities:

* Insertion of keywords and meanings
* Searching for a keyword and displaying its meaning
* Modifying the meaning of a keyword
* Deletion of a keyword
* Various tree traversals (preorder, inorder, postorder, ascending, descending)
* Counting the number of comparisons during search

**✅ Function-by-Function Explanation**

**🔹 node Class**

This class represents the structure of a node in the AVL tree.

* **Attributes:**
  + keyword: Stores the keyword (word).
  + meaning: Stores the meaning of the keyword.
  + left: Pointer to the left child node.
  + right: Pointer to the right child node.
  + ht: Height of the node (used for balancing).

**🔹 tree Class**

This class contains the AVL tree and its operations.

**1. height(node \*T)**

**Purpose:** Returns the height of the node T.  
**Algorithm:**

* If the node is NULL, return 0.
* Otherwise, calculate the height of both left and right subtrees and return the maximum of the two plus one.

**Time Complexity:** O(1)  
(Returning the height of a node is a constant-time operation.)

**2. BF(node \*T)**

**Purpose:** Returns the balance factor of the node T, which helps determine if the tree is balanced.  
**Algorithm:**

* Balance factor is calculated as the difference between the heights of the left and right subtrees.

**Time Complexity:** O(1)  
(Just performing arithmetic operations and accessing the height attributes.)

**3. Rotate\_Right(node \*x)**

**Purpose:** Performs a right rotation on the node x to maintain balance.  
**Algorithm:**

* Perform the right rotation and update the heights of the affected nodes.

**Time Complexity:** O(1)  
(Only a few pointer assignments and height updates are needed.)

**4. Rotate\_Left(node \*T)**

**Purpose:** Performs a left rotation on the node T to maintain balance.  
**Algorithm:**

* Perform the left rotation and update the heights of the affected nodes.

**Time Complexity:** O(1)  
(Similar to the right rotation.)

**5. insert(node \*T, char word[20], char mean[20])**

**Purpose:** Inserts a keyword and its meaning into the AVL tree while maintaining balance.  
**Algorithm:**

* If the tree is empty, create a new node.
* Compare the new keyword with the current node's keyword to decide whether to insert in the left or right subtree.
* If insertion causes the tree to become unbalanced, perform appropriate rotations (right or left).

**Time Complexity:** O(log n)  
(Each insertion involves comparing keywords and performing rotations, which takes logarithmic time due to the balanced nature of the AVL tree.)

**6. insert1(char key[20], char mean[20])**

**Purpose:** A wrapper for the insert function that calls insert on the root node.  
**Time Complexity:** O(log n)  
(Similar to the insert function.)

**7. Tree Traversals (preorder, inorder, postorder, descending)**

**Purpose:** These functions traverse the tree in different orders:

* **Preorder**: Root, Left, Right
* **Inorder**: Left, Root, Right
* **Postorder**: Left, Right, Root
* **Descending**: Right, Root, Left (opposite of inorder)

**Time Complexity:** O(n)  
(Each traversal visits every node once.)

**8. deleteNode(node \*root, char key[20])**

**Purpose:** Deletes a keyword and its meaning from the tree while maintaining balance.  
**Algorithm:**

* If the node is a leaf or has only one child, delete it directly.
* If the node has two children, find the inorder successor (smallest node in the right subtree), replace the current node with the inorder successor, and then delete the successor.
* After deletion, check and balance the tree using rotations if necessary.

**Time Complexity:** O(log n)  
(Finding and deleting a node, as well as balancing the tree, can be done in logarithmic time.)

**9. modify(node \*T)**

**Purpose:** Modifies the meaning of a given keyword in the tree.  
**Algorithm:**

* Traverse the tree to find the node with the given keyword.
* If found, modify the meaning. Otherwise, print "Word not found."

**Time Complexity:** O(log n)  
(Searching for the word takes logarithmic time in an AVL tree.)

**10. search1(node \*T)**

**Purpose:** Searches for a keyword in the tree and prints its meaning if found.  
**Time Complexity:** O(log n)  
(Similar to modify, searching for a word in an AVL tree takes logarithmic time.)

**11. max\_compare(node \*T)**

**Purpose:** Searches for a keyword in the tree and counts the number of comparisons made during the search.  
**Time Complexity:** O(log n)  
(Similar to search1, but with an additional comparison count.)

**12. display()**

**Purpose:** Displays the main menu and allows the user to choose from various operations like insert, search, modify, delete, etc.  
**Time Complexity:** O(1)  
(The menu is displayed in constant time.)

**🧾 Summary Table**

| **Function** | **Purpose** | **Time Complexity** |
| --- | --- | --- |
| height(node \*T) | Returns the height of a node | O(1) |
| BF(node \*T) | Returns the balance factor of a node | O(1) |
| Rotate\_Right(node \*x) | Performs a right rotation | O(1) |
| Rotate\_Left(node \*T) | Performs a left rotation | O(1) |
| insert(node \*T, char\*, char\*) | Inserts a node and balances the tree | O(log n) |
| insert1(char\*, char\*) | Wrapper for the insert function | O(log n) |
| preorder(node \*T) | Preorder traversal | O(n) |
| inorder(node \*T) | Inorder traversal | O(n) |
| postorder(node \*T) | Postorder traversal | O(n) |
| descending(node \*T) | Descending order traversal | O(n) |
| deleteNode(node\*, char\*) | Deletes a node and rebalances the tree | O(log n) |
| modify(node \*T) | Modifies a keyword's meaning | O(log n) |
| search1(node \*T) | Searches for a keyword | O(log n) |
| max\_compare(node \*T) | Counts comparisons during a search | O(log n) |
| display() | Displays the main menu | O(1) |

**📝 Conclusion**

* This program efficiently implements an AVL tree-based dictionary that ensures fast operations even as the number of entries grows.
* Operations like insertion, deletion, and searching are handled in **logarithmic time** due to the self-balancing nature of the AVL tree.
* The use of tree traversals and the ability to modify, search, and delete entries makes this a fully functional dictionary application.

**UNDERSTANDING-BASED QUESTIONS (CONCEPTUAL)**

1. **What is a heap?**
   * A heap is a complete binary tree where each node follows a specific order property. It can be either a **Min Heap** or a **Max Heap**.
2. **What is the difference between Min Heap and Max Heap?**
   * **Min Heap:** Parent node is always smaller than or equal to its children.
   * **Max Heap:** Parent node is always greater than or equal to its children.
3. **What is meant by "heapify"?**
   * "Heapify" is the process of adjusting a binary tree to satisfy the heap property from a given node down to its children.
4. **Why do we start heapify from n/2 to 1 in heap building?**
   * Nodes from n/2 + 1 to n are leaf nodes and already satisfy heap property. So, we start heapifying from the last non-leaf node up to the root.
5. **What is the time complexity of building a heap?**
   * Time complexity is **O(n)** for building a heap using the bottom-up approach.
6. **Is heap a binary search tree (BST)?**
   * No, a heap is not a BST. BST follows left < root < right; heap only follows parent-child ordering.
7. **Can heap be used for sorting?**
   * Yes, **Heap Sort** algorithm uses a heap to sort elements in **O(n log n)** time.
8. **What are real-life applications of heaps?**
   * **Priority queues**, **job scheduling**, **Dijkstra’s algorithm**, **Heap Sort**, etc.

**🧠 CODE-SPECIFIC QUESTIONS**

1. **Why are the array indices starting from 1 instead of 0?**
   * It simplifies the formula:
     + Left child: 2 \* i
     + Right child: 2 \* i + 1
     + Parent: i / 2
2. **What does the function min\_heapify() do?**
   * It ensures the subtree rooted at index i satisfies the Min Heap property.
3. **Why do we use temp = a[i] in heapify functions?**
   * It stores the original value being compared and shifted down to avoid repeated swaps.
4. **What happens if we enter a non-complete binary tree?**
   * The heap requires the tree to be complete. The input array simulates a complete binary tree.
5. **What happens when j > n in heapify?**
   * The while loop exits since the child node is out of bounds.
6. **What if all elements are equal?**
   * Both Min and Max Heap properties will be satisfied since every node equals its children.
7. **What does a[j/2] = temp; mean at the end?**
   * It places the stored value in its correct position in the heap.

**📈 OUTPUT-BASED / DRY-RUN QUESTIONS**

1. **Given an input array, can you manually show how it converts to a Min Heap?**
2. **If input is: 50, 30, 20, 15, 10, 8, 16**, show how the Max Heap looks.
3. **Show the steps of heapify for index i = 2 in a Min Heap.**

**🧪 TRICKY / CRITICAL-THINKING QUESTIONS**

1. **Why is heap not stable like Merge Sort?**
   * Heap doesn’t preserve the order of equal elements.
2. **What would happen if we removed the break statement in heapify?**
   * It could cause unnecessary comparisons or even infinite loops.
3. **Why is recursion not used in heapify here?**
   * Iterative heapify is more space-efficient as it avoids stack overflow for deep trees.
4. **Can we build a heap with fewer comparisons?**
   * The bottom-up heapify already provides a linear time build with minimal comparisons.
5. **Can we directly sort the array after building a heap?**
   * Not directly with this code; you need to implement heap sort logic to extract elements one by one.

**🧾 CODE IMPROVEMENT / EXTENSION QUESTIONS**

1. **How would you extend this code to perform Heap Sort?**
2. **Can you modify this to implement a priority queue?**
3. **What changes would be required to support dynamic insertions and deletions?**
4. **Can you implement the same using classes instead of functions?**

ASSIGNMENT 11

This C++ program provides functionality to manage student records in a binary file (stud.dat). It allows you to **create**, **display**, **search**, and **delete** student records. The records consist of basic student information: roll number, name, division, and address.

**🛠 Code Explanation**

**1. student class:**

This class manages student records, and it has the following member functions:

* **create()**: Adds new student records to the file.
* **display()**: Displays all the student records stored in the file.
* **search()**: Searches for a student record based on the roll number.
* **Delete()**: Deletes a student record by marking it as deleted (using roll = -1).

**2. create():**

* This function allows the user to enter student details (roll number, name, division, address).
* It writes these details to a binary file (stud.dat).
* The user can choose to add more records using a loop (y or Y to continue).

**3. display():**

* This function reads the student records from the binary file (stud.dat) and displays them in a table format.
* The roll number is checked to ensure the record is not marked as deleted (roll != -1).

**4. search():**

* The function searches for a student by their roll number.
* If a matching record is found, it displays the record's details.
* If the student is not found, it returns 0.

**5. Delete():**

* This function first searches for the record by roll number.
* If the record is found, it marks the student as deleted by setting roll = -1 and replacing the name and address with "NULL".
* It writes the updated record back to the file.

**6. Main Menu (main()):**

* A loop allows the user to choose between creating, displaying, deleting, or searching student records.
* The program continues prompting the user until they choose to exit.

**🧮 Time Complexity Analysis**

* **create()**: Each insertion of a record takes O(1) time for writing the record to the file.
* **display()**: It requires reading all student records from the file, which takes O(n) time where n is the number of students.
* **search()**: The search function performs a linear scan of the file, so its time complexity is O(n) in the worst case.
* **Delete()**: The delete operation involves searching for a record (O(n)) and then modifying it (O(1) for marking as deleted).

**📊 Example Execution**

**Sample Input (for creating student records):**

Enter Roll No of Student: 1

Enter a Name of Student: John

Enter a Division of Student: A

Enter a Address of Student: 123 St

Do You Want to Add More Records: y

Enter Roll No of Student: 2

Enter a Name of Student: Jane

Enter a Division of Student: B

Enter a Address of Student: 456 Ave

Do You Want to Add More Records: n

**Menu Options:**

1. Create
2. Display
3. Delete
4. Search
5. Exit

**Output for Displaying Records:**

The Content of File are:

Roll Name Div Address

1 John A 123 St

2 Jane B 456 Ave

**Output for Searching:**

Enter a Roll No: 1

Record Found...

Roll Name Div Address

1 John A 123 St

**Output for Deleting:**

Enter a Roll No: 1

Record Deleted

ASSIGNMENT 12

This C++ program defines a class EMP\_CLASS to manage employee records in a system with a sequential file (EMP.DAT) for storing employee data and an index file (IND.DAT) for indexing employee records based on their emp\_id. It provides functionality for creating, displaying, updating, deleting, appending, and searching employee records.

**🛠 Code Explanation**

**1. EMP\_CLASS Class Structure:**

* **EMPLOYEE Structure**: Defines the data structure for employee records with fields:
  + emp\_id: Employee ID
  + name: Employee name (stored as a fixed-size array of 10 characters)
  + salary: Employee salary
* **INDEX Structure**: Used for indexing, it stores:
  + emp\_id: Employee ID
  + position: Position in the sequential file (EMP.DAT)

**2. Functions in EMP\_CLASS:**

* **Create()**:
  + This function allows the user to create employee records. It writes both the employee data (EMP.DAT) and the index data (IND.DAT).
  + The user can input employee details, and the data is written to the respective files.
  + It prompts the user to continue adding more records.
* **Display()**:
  + This function reads the index file (IND.DAT) to get positions of records in the sequential file (EMP.DAT), and then it reads the corresponding employee records and displays them.
* **Update()**:
  + Allows the user to update an existing employee's details by searching using the emp\_id.
  + If the record is found, it updates the employee's name and salary in the sequential file (EMP.DAT).
* **Delete()**:
  + This function deletes an employee record by setting the employee data to "empty" (with emp\_id = -1 and empty fields).
  + It also deletes the corresponding index record from the index file.
* **Append()**:
  + This function appends a new employee record to the end of both the sequential file (EMP.DAT) and the index file (IND.DAT).
  + It writes the employee data to both files.
* **Search()**:
  + This function allows searching for an employee by their emp\_id.
  + It reads the index file (IND.DAT) to find the employee's position in the sequential file and then reads and displays the employee record.

**3. Main Menu (main()):**

* A menu system is used to choose between creating, displaying, updating, deleting, appending, and searching employee records.
* The program loops through the menu until the user chooses to exit.

**🧮 Time Complexity Analysis**

* **Create()**: Writing to both the sequential file and the index file takes O(1) time for each employee record. The overall complexity depends on the number of records added.
* **Display()**: This function reads both files sequentially, resulting in an overall complexity of O(n), where n is the number of records.
* **Update()**: Searching for the record takes O(n) because it reads through the index file. After finding the record, the update operation itself is O(1).
* **Delete()**: Similar to the update function, the delete operation involves searching (O(n)), followed by writing a modified record (O(1)).
* **Append()**: Appending is an O(1) operation, but it requires updating the index file and the sequential file.
* **Search()**: Searching is O(n) because it reads through the index file to find the corresponding record.

**📊 Example Execution**

**Sample Input (for creating records):**

Enter Emp\_ID: 101

Enter Name: John

Enter Salary: 50000

Do you want to add more records? (y/n): y

Enter Emp\_ID: 102

Enter Name: Jane

Enter Salary: 55000

Do you want to add more records? (y/n): n

**Menu Options:**

1. Create
2. Display
3. Update
4. Delete
5. Append
6. Search
7. Exit

**Output for Displaying Records:**

The Contents of file are:

Name : John

Emp\_ID : 101

Salary : 50000

Name : Jane

Emp\_ID : 102

Salary : 55000

**Output for Searching:**

Enter the Emp\_ID to search: 101

Record found:

Name : John

Emp\_ID : 101

Salary : 50000

**Output for Updating:**

Enter the Emp\_ID to update: 101

Enter new Name: John Doe

Enter new Salary: 60000

Record updated successfully!

**Output for Deleting:**

Enter the Emp\_ID to delete: 101

Record deleted successfully!

**Output after Displaying Again:**

The Contents of file are:

Name : John Doe

Emp\_ID : -1

Salary : -1

Name : Jane

Emp\_ID : 102

Salary : 55000