C15(connect offices:min cost)

1. **Class `Office`**: Represents an office network. It contains the following members:
    - `n`: Number of offices.
    - `adjacent[10][10]`: Adjacency matrix to store the cost of connecting offices.
    - `office[10]`: Array to store the names of offices.
2. **Function `input()`**:
    - Takes input for the number of offices (`n`) and their names (`office` array).
    - Takes input for the cost of connecting each pair of offices and stores it in the adjacency matrix (`adjacent`).
3. **Function `display()`**:
    - Displays the adjacency matrix, showing the cost of connecting each pair of offices.
4. **Function `Prims()`**:
    - Implements Prim's algorithm to find the Minimum Spanning Tree (MST).
    - Initializes an array `visit[]` to keep track of visited offices and sets all offices as unvisited initially.
    - Marks the first office as visited.
    - Iterates `n-1` times (as the MST will have `n-1` edges).
    - Finds the minimum cost edge connecting a visited office to an unvisited office.
    - Marks the unvisited office as visited and adds the cost of the edge to the total cost of the MST.
5. **Main Function**:
    - Creates an object of class `Office`.
    - Displays a menu-driven interface to:
        - Input data (number of offices, names, and connection costs).
        - Display the adjacency matrix.
        - Calculate and display the MST using Prim's algorithm.

E22 HEAP SORT MAX HEAP,MIN HEAP

1. **Class `Heap_op`**: This class is designed to handle operations related to max and min heaps for student marks.
2. **Function `create()`**:
    - This function initializes the max and min heaps as empty arrays and then prompts the user to enter the number of students (`n`).
    - It then iterates `n` times, prompting the user to enter the mark of each student (`x`), and inserts these marks into both the max and min heaps using the `max_insert()` and `min_insert()` functions respectively.
3. **Function `max_insert(int maxHeap[], int data)`**:
    - This function inserts a new element `data` into the max heap `maxHeap[]`.
    - It first increments the heap size (`maxHeap[0]`) and then inserts `data` at the end of the heap.
    - After insertion, it performs an up-adjustment (`up_adjust()`) starting from the newly inserted element to maintain the max heap property.
4. **Function `up_adjust(int maxHeap[], int i)`**:
    - This function performs an up-adjustment starting from the index `i` in the max heap `maxHeap[]`.
    - It compares the element at index `i` with its parent (`i/2`), and if the element is greater than its parent, it swaps them.
    - This process continues until the element is in its correct position in the heap or until it reaches the root.
5. **Function `display_max()`**:
    - This function displays the highest mark in the max heap (which is the root of the max heap).
    - It then displays the elements of the max heap in their current order, which represents the traversal of the max heap tree from left to right, top to bottom.
6. **Function `min_insert(int minHeap[], int data)` and Function `down_adjust(int minHeap[], int i)`**:
    - These functions are similar to `max_insert()` and `up_adjust()`, but they handle the min heap instead of the max heap.
    - `min_insert()` inserts an element into the min heap and performs a down-adjustment (`down_adjust()`) to maintain the min heap property.
7. **Function `display_min()`**:
    - This function displays the lowest mark in the min heap (which is the root of the min heap).
    - It then displays the elements of the min heap in their current order, representing the traversal of the min heap tree from left to right, top to bottom.
8. **Main Function**:
    - Creates an object `obj` of class `Heap_op` and calls its `create()` function to initialize the heaps and input student marks.

- Calls `display_max()` and `display_min()` to display the highest and lowest marks along with the traversal of the max and min heaps respectively.

B5(book consists of chapters, chapters consist of sections)

1. **Struct `node`**:
    - Represents a node in the hierarchical structure.
    - Contains two members: `label` (string) to store the name of the node, and `count` (integer) to store the number of children nodes.
    - Also contains an array `child` of pointers to `node`, to store the child nodes.
2. **Class `Tree`**:
    - Contains two public member functions: `insert()` and `display()`.
    - Constructor initializes the `root` pointer to `NULL`.
3. **Function `insert()`**:
    - Creates the root node of the tree and prompts the user to enter the name of the book and the total number of chapters.
    - For each chapter, it creates a new node, prompts for the name of the chapter, and the number of sections.
    - For each section, it creates a new node, prompts for the name of the section, and the number of subsections.
    - For each subsection, it creates a new node and prompts for the name of the subsection.
4. **Function `display()`**:
    - Checks if the `root` node is not `NULL`.
    - Displays the book name (root node's `label`).
    - Iterates over each chapter, displaying the chapter name and then iterating over each section in the chapter, displaying the section name and then iterating over each subsection in the section, displaying the subsection name.
5. **Main Function**:
    - Creates an instance `g` of class `Tree`.
    - Displays a menu for the user to choose between inserting nodes and displaying the tree structure.
    - Loops until the user chooses to exit.

C13 (perform DFS and using adjacency list to perform BFS.)

1. **Class `Graph`**:
   - Contains private variables `n` (number of vertices), `e` (number of edges), and `adjMat` (adjacency matrix).
   - Provides methods to create a graph using an adjacency matrix (`createUsingAdjMat()`), display the adjacency matrix (`displayAdjMat()`), perform DFS traversal (`dfs()`), and perform BFS traversal (`bfs()`).
2. **Constructor**:
   - Initializes the adjacency matrix `adjMat` to all zeros.
3. **Method `createUsingAdjMat()`**:
   - Takes input for the number of vertices `n` and edges `e`.
   - Initializes the adjacency matrix `adjMat` with zeros.
   - Takes input for each edge and updates the adjacency matrix accordingly.
4. **Method `displayAdjMat()`**:
   - Displays the adjacency matrix `adjMat`.
5. **Method `dfs(int start)`**:
   - Performs depth-first search traversal starting from the vertex `start`.
   - Uses a stack to keep track of vertices to visit.
   - Marks visited vertices to avoid revisiting them.
6. **Method `bfs(int start)`**:
   - Performs breadth-first search traversal starting from the vertex `start`.
   - Uses a queue to keep track of vertices to visit.
   - Marks visited vertices to avoid revisiting them.
7. **Main Function**:
   - Creates an instance `g` of class `Graph`.
   - Calls `createUsingAdjMat()` to create the graph.
   - Calls `displayAdjMat()` to display the adjacency matrix.
   - Takes input for the starting vertex for DFS traversal and calls `dfs()` with this vertex.
   - Takes input for the starting vertex for BFS traversal and calls `bfs()` with this vertex.

D18 (OBST)

1. **Class `OBST`**:
    - Contains private member variables `prob` (probabilities), `keys` (keys to be inserted in the tree), `weight` (weights of subtrees), `cost` (cost of optimal BSTs), `root` (root of optimal BSTs), and `n` (number of nodes).
    - Provides methods to get data (`get_data()`), find the minimum value (`Min_Value()`), build the OBST (`build_OBST()`), build the tree (`build_tree()`), and print the tables (`print()`).
2. **Method `get_data()`**:
    - Takes input for the number of nodes, keys, and probabilities.
3. **Method `Min_Value(int i, int j)`**:
    - Finds the key value in the range `r[i][j-1]` to `r[i+1][j]` so that the cost `cost[i][k-1]+cost[k][j]` is minimum.
4. **Method `build_OBST()`**:
    - Initializes the weight, cost, and root tables for all possible subtrees.
    - Computes the cost, root, and weight values for each subtree using dynamic programming.
5. **Method `build_tree()`**:
    - Builds the optimal BST using the root table computed in `build_OBST()`.
6. **Main Function**:
    - Creates an instance `obj` of class `OBST`.
    - Calls `get_data()` to get input.
    - Calls `build_OBST()` to build the OBST.
    - Calls `build_tree()` to build the tree.
    - Displays the root and cost of the optimal BST.

B6 Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree - i. Insert new node, ii. Find number of nodes in longest path from root, iii. Minimum data value found in the tree, iv. Change a tree so that the roles of the left and right pointers are swapped at every node, v. Search a value .

The code begins with the inclusion of necessary libraries and the definition of a Node struct to represent a node in a binary tree. Each node has an integer data value (data) and pointers to its left and right children (left and right).

Next, a Tree class is defined. This class represents a binary search tree (BST). It contains a private member root which points to the root of the binary tree.

The constructor Tree() initializes the root pointer to NULL, indicating an empty tree.

The createNode() function dynamically allocates memory for a new node, initializes its data value, and sets its left and right pointers to NULL.

The insertNode() function recursively inserts a new node with the given data into the tree at the appropriate position based on the binary search tree property (nodes in the left subtree are less than the root, and nodes in the right subtree are greater than the root).

The insert() function inserts a new node into the tree by calling insertNode().

The inorder() function performs an inorder traversal of the tree, printing the data of each node in sorted order.

The display() function prints the contents of the tree by calling inorder().

The search_data() function searches for a given key in the tree using a recursive approach.

The search() function initiates the search for a given data value in the tree.

The minimum() function finds the smallest data value in the tree by traversing to the leftmost node.

The swap() function recursively swaps the left and right subtrees of each node in the tree, effectively creating a mirror image of the tree.

The mirror() function initiates the process of mirroring the tree by calling swap().

The maximum() function returns the maximum of two integers.

The height_node() function calculates the height of a node in the tree recursively.

The height() function calculates the height of the tree by calling height_node().

In the main() function, an instance of the Tree class is created.

A loop is used to display a menu and perform various operations on the binary search tree based on user input.

The user can choose to insert data into the tree, display the contents of the tree, mirror the tree, find the maximum height of the tree, find the smallest data value in the tree, search for a specific data value, or exit the program.

The loop continues until the user chooses to exit by entering option 7.

A4 To create ADT that implement the "set" concept. a. Add (new Element) -Place a value into the set , b. Remove (element) Remove the value c. Contains (element) Return true if element is in collection, d. Size () Return number of values in collection Iterator () Return an iterator used to loop over collection, e. Intersection of two sets , f. Union of two sets, g. Difference between two sets, h. Subset

1. **Header Inclusion**: The code includes the necessary header file `<iostream>` for input-output operations.
2. **Namespace**: The `using namespace std;` statement brings all elements of the standard C++ library into the current namespace, allowing you to use `cout`, `cin`, etc., without prefixing them with `std::`.
3. **Template Class Declaration**: The code defines a template class named `set`. This class is intended to represent a mathematical set.
4. **Private Data Members**:
   - `max`: Represents the maximum size of the set.
   - `table`: Represents the array to hold elements of the set.
5. **Constructor**:
   - Initializes `max` to 0.
   - Dynamically allocates memory for the `table` array using the `new` keyword.
6. **Add Method**: Adds an element to the set if it's not already present.
7. **Remove Method**: Removes an element from the set.
8. **Display Method**: Displays the elements of the set.
9. **Size Method**: Returns the current size of the set.
10. **Element Method**: Returns the element at a specified index in the set.
11. **Intersection Method**: Finds the intersection of two sets and displays the result.
12. **Union Method**: Finds the union of two sets and displays the result.
13. **Subset Method**: Checks if one set is a subset of another or if they are equal sets.
14. **Difference Method**: Finds the difference between two sets (A - B or B - A) and displays the result.
15. **Main Function**:
    - Creates two sets `a` and `b`.
    - Allows the user to add, remove, and display elements in both sets.
    - Performs set operations like intersection, union, difference, and subset checking based on user input.
16. **Menu-driven User Interface**:
    - Allows users to interactively perform set operations.
    - Loops until the user chooses to exit.

VOID ADD :

This `Add` method in the `set` class is responsible for adding an element to the set. Let's break down its functionality:

1. It takes a parameter `a` of type `T`, representing the element to be added to the set.
2. It initializes a flag `f` to 0. This flag will be used to determine whether the element already exists in the set.
3. It iterates through the elements of the set stored in the `table` array using a for loop.
4. Within the loop, it checks if the current element (`table[j]`) is equal to the element `a` being added. If they are equal, it sets the flag `f` to 1 and breaks out of the loop.
5. After the loop, it checks the value of the flag `f`. If `f` is 1, it means that the element already exists in the set, so it simply returns without adding the element again.
6. If `f` is 0, it means the element does not exist in the set. In this case, it creates a temporary array `temp` of size `max + 1` using dynamic memory allocation (`new`).
7. It then copies all elements from the current `table` array to the `temp` array.
8. It adds the new element `a` to the end of the `temp` array.
9. It deallocates the memory occupied by the current `table` array using `delete[]`.
10. It assigns the address of the `temp` array to the `table` pointer, effectively updating the `table` array to point to the new array with the added element.
11. It increments the `max` variable to reflect the increased size of the set.
12. Finally, it returns, indicating that the addition operation is complete.

VOID REMOVE :

This `remove` method in the `set` class is responsible for removing an element from the set. Here's how it works:

1. It takes a parameter `a` of type `T`, representing the element to be removed from the set.
2. It iterates through the elements of the set stored in the `table` array using a for loop.

3. Within the loop, it checks if the current element (`table[j]`) is equal to the element `a` being removed. If they are equal, it proceeds to remove the element.
4. Inside the `if` condition, it initializes a variable `u` to 0, which will be used as an index offset to shift elements in the array.
5. It enters a `while` loop that continues until `u` becomes equal to `max`. This loop shifts all elements after the one to be removed one position to the left, effectively removing the element.
6. After shifting the elements, it decrements the `max` variable to reflect the reduced size of the set.
7. It then returns from the method, indicating that the removal operation is complete.
8. If the element to be removed is not found in the set (i.e., the loop completes without finding a matching element), it prints a message indicating that the element was not found.

This method effectively removes an element from the set by shifting all elements after it in the array. If the element is not found, it notifies the user.

INTERSECTION :

This `Intersection` method in the `set` class is responsible for finding the intersection of two sets `a` and `b`. Here's how it works:

1. It takes two parameters `a` and `b`, which are references to instances of the `set` class representing the sets to find the intersection of.
2. It initializes a new set `c`, which will hold the intersection of sets `a` and `b`.
3. It iterates through the elements of set `a` using a for loop.
4. Inside the loop, it iterates through the elements of set `b` using another nested for loop.
5. Within the nested loop, it checks if the current element of set `a` (`a.element(i)`) is equal to any element of set `b` (`b.element(j)`).
6. If a matching element is found, it adds that element to the intersection set `c` using the `Add` method and then breaks out of the inner loop to move to the next element of set `a`.
7. After both loops complete, set `c` contains all the elements that are common to both sets `a` and `b`, i.e., the intersection of the two sets.
8. The method does not return anything, but the intersection set `c` is available for further use.

UNION

This `Union` method in the `set` class is responsible for finding the union of two sets `a` and `b`. Here's how it works:

1. It takes two parameters `a` and `b`, which are references to instances of the `set` class representing the sets for which the union is to be found.
2. It initializes a new set `c`, which will hold the union of sets `a` and `b`.
3. It iterates through the elements of set `a` using a for loop.
4. Inside the loop, it iterates through the elements of set `b` using another nested for loop.
5. Within the nested loop, it checks if the current element of set `a` (`a.element(i)`) is not equal to the current element of set `b` (`b.element(j)`).
6. If the elements are not equal, it adds both elements to the union set `c` using the `Add` method. This ensures that all unique elements from both sets are included in the union.
7. If the elements are equal, it adds only one of the elements to the union set `c`. This prevents duplicate elements from being added to the union.
8. After both loops complete, set `c` contains all the unique elements from sets `a` and `b`, forming the union of the two sets.
9. Finally, it displays the elements of the union set `c` using the `display` method.
10. The method returns after displaying the union set.

SUBSET :

This `Subset` method in the `set` class is responsible for determining whether one set is a subset of another. Here's how it works:

1. It takes two parameters `a` and `b`, which are references to instances of the `set` class representing the sets to be compared.
2. It initializes a variable `count` to 0. This variable will count the number of elements in set `a` that are also present in set `b`.
3. It iterates through the elements of set `a` using a for loop.
4. Inside the outer loop, it iterates through the elements of set `b` using another nested for loop.
5. Within the nested loop, it checks if the current element of set `a` (`a.element(i)`) is equal to the current element of set `b` (`b.element(j)`).
6. If a matching element is found, it increments the `count` variable.
7. After both loops complete, the `count` variable contains the total number of elements in set `a` that are also present in set `b`.

8. It then compares the `count` variable with the size of set `a`. If `count` is equal to the size of set `a`, it means all elements of set `a` are present in set `b`, indicating that set `a` is a subset of set `b`.
9. Depending on the comparison result, it prints appropriate messages indicating whether set `a` is a subset of set `b`, set `b` is a subset of set `a`, both sets are subsets of each other, or none of the sets is a subset of the other.

DIFFERENCE :

This `Difference` method in the `set` class is responsible for finding the difference between two sets `a` and `b`. It allows the user to choose between finding the difference A - B or B - A. Here's how it works:

1. It takes two parameters `a` and `b`, which are references to instances of the `set` class representing the sets for which the difference is to be found.
2. It initializes a new set `c`, which will hold the result of the difference operation.
3. It prompts the user to choose between two options: A - B or B - A.
4. It reads the user's choice using `cin`.
5. It uses a `switch` statement to handle the user's choice:
    - If the choice is 1 (A - B), it iterates through the elements of set `a` using a for loop.
        - Inside the loop, it checks if the current element of set `a` is not present in set `b`.
        - If the element is not present in set `b`, it adds it to the difference set `c`.
        - After the loop completes, it displays the elements of set `c` using the `display` method, showing the difference A - B.
    - If the choice is 2 (B - A), it iterates through the elements of set `b` using a for loop.
        - Inside the loop, it checks if the current element of set `b` is not present in set `a`.
        - If the element is not present in set `a`, it adds it to the difference set `c`.
        - After the loop completes, it displays the elements of set `c` using the `display` method, showing the difference B - A.

B11 A Dictionary stores keywords and its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Binary Search Tree for implementation

1. `#include <iostream>`: Includes the Input/Output Stream Library for C++.
2. `using namespace std;`: Declares that all identifiers in the `std` namespace are available for unqualified use.
3. `class node { ... };`: Defines a class named `node`, representing nodes in a binary search tree. It contains two string variables `word` and `meaning` to store the word and its meaning. It also has two pointers `left` and `right` to point to the left and right child nodes, respectively. A constructor is defined to initialize the `word`, `meaning`, `left`, and `right` variables.
4. `friend class Dictionary;`: Declares the `Dictionary` class as a friend of the `node` class, allowing `Dictionary` class to access the private members of `node`.
5. `class Dictionary { ... };`: Defines a class named `Dictionary`, which represents a binary search tree-based dictionary. It contains private member variables `root` and `q`, which are pointers to nodes in the tree. A constructor initializes both pointers to `NULL`. It also contains several member functions for dictionary operations.
    6. `void Dictionary::insert (node* p, string key, string keyMeaning) { ... }`: This is the declaration of the `insert` function. It takes three parameters:
        1. `p`: A pointer to the current node in the binary search tree.
        2. `key`: The word to be inserted into the tree.
        3. `keyMeaning`: The meaning of the word to be inserted.
    7. `if (key < p -> word) { ... }`: This condition checks if the `key` should be inserted in the left subtree of the current node `p` based on the lexicographical order of `key` and the word stored in the current node `p`.
    8. `if (p -> left != NULL) insert (p -> left, key, keyMeaning);`: If the left child of `p` is not `NULL`, it recursively calls the `insert` function with the left child of `p` as the current node.
    9. `else p -> left = new node (key, keyMeaning);`: If the left child of `p` is `NULL`, it creates a new node with the `key` and `keyMeaning` and assigns it as the left child of `p`.
    10. `else if (key > p -> word) { ... }`: If `key` is greater than the word stored in the current node `p`, it checks if the right child of `p` is not `NULL`.
    11. `if (p -> right != NULL) insert (p -> right, key, keyMeaning);`: If the right child of `p` is not `NULL`, it recursively calls the `insert` function with the right child of `p` as the current node.

12. `else p -> right = new node (key, keyMeaning);`: If the right child of `p` is `NULL`, it creates a new node with the `key` and `keyMeaning` and assigns it as the right child of `p`.
13. `void Dictionary::display_asc(node *p)`: This function displays the binary search tree in ascending order.
    1. It takes a pointer to the root node of the binary search tree as input.
    2. It recursively traverses the left subtree first (`if (p->left != NULL) display_asc(p->left)`).
    3. Then, it prints the word and its meaning stored in the current node (`cout << "\n" << p->word << " \t" << p->meaning`).
    4. Finally, it recursively traverses the right subtree (`if (p->right != NULL) display_asc(p->right)`).
14. `void Dictionary::display_desc(node *p)`: This function displays the binary search tree in descending order.
    1. Similar to `display_asc`, it starts by recursively traversing the right subtree first (`if (p->right != NULL) display_desc(p->right)`).
    2. Then, it prints the word and its meaning stored in the current node (`cout << "\n" << p->word << " \t" << p->meaning`).
    3. Finally, it recursively traverses the left subtree (`if (p->left != NULL) display_desc(p->left)`).
15. `static int count = 0;`: This line declares a static integer variable `count` and initializes it to 0. Static variables retain their values between function calls.
16. `while (p != NULL) {`: This initiates a loop that continues until the pointer `p` reaches a null value, indicating the end of the search path.
17. Inside the loop:
    1. `if (key < p->word) {`: Checks if the key (word being searched) is less than the word stored in the current node. If true, it increments the `count` and moves the pointer `p` to the left child node (`p = p->left;`), continuing the search in the left subtree.
    2. `else if (key > p->word) {`: Checks if the key is greater than the word stored in the current node. If true, it increments the `count` and moves the pointer `p` to the right child node (`p = p->right;`), continuing the search in the right subtree.
    3. `else if (key == p->word) {`: Checks if the key matches the word stored in the current node. If true, it increments the `count`, prints the number of comparisons, and returns from the function.
18. If the loop exits without finding the word (`p` becomes `NULL`), it means the word was not found, and it prints a message indicating that.
19. `updateWord`:

    1. This function is responsible for updating the meaning of a word in the dictionary.
    2. It takes a pointer to the current node `p` and the key (word) to be updated.
    3. It traverses the binary search tree until it finds the node containing the given key or until it reaches a leaf node (NULL), similar to the `comparisons` function.
    4. If the key is found (`key == p->word`), it prompts the user to enter the new meaning for the word, which is then stored in the `meaning` member variable of the node.
    5. If the key is not found, it prints a message indicating that the word was not found in the dictionary.

20. `min_node`:
    1. This function is used to find the node with the minimum key (word) value in a subtree.
    2. It takes a pointer to the root of the subtree `p`.
    3. It traverses the left child nodes of the subtree until it reaches the leftmost node, which has the smallest key value.
    4. During this traversal, it updates a reference pointer `q` to keep track of the parent node of the minimum node.
    5. Once it reaches the leftmost node, it returns a pointer to that node.

Both of these functions are essential for various operations on the binary search tree, such as updating a word's meaning and finding the minimum node (which is often used in deletion operations).

F23. Department maintains a student information. The file contains roll number, name, division and address. Allow user to add, delete information of student. Display information of particular employee. If record of student does not exist an appropriate message is displayed. If it is, then the system displays the student details. Use sequential file to main the data.

Sure, let's break down the code line by line:

cpp
Copy code
```
include<iostream> include<fstream> include<string.h> using namespace
```

- These lines include necessary header files for input/output operations and file handling.

cpp
Copy code
```
typedef struct stud int char 10 char char 10
```

- This defines a structure `stud` to hold student records, including roll number, name, division, and address.

cpp
Copy code
```
class student public void insert void display int search void Delete
getdata
```

- This declares a class `student` with member functions to insert, display, search, delete records, and a helper function to get data from the user.

cpp
Copy code
```
student::getdata "\n Enter roll no: " "\n Enter name: "
"\n Enter div: " "\n Enter add: " return
```

- This function gets data from the user for a student record and returns the `stud` structure.

Let's break down the `student::insert` function line by line:

cpp
Copy code
```
void student::insert int
```

- This function takes a `stud` object `rec1` as an argument. It also declares some integer variables `i`, `n`, and `k` along with a local `stud` object `rec`.

cpp
Copy code
```
open "stud.dat"
```

- This line declares and opens a file stream object `f` for handling the file named "stud.dat". It opens the file in both output and input modes, enabling both reading and writing operations. The mode `ios::binary` indicates that the file will be treated as a binary file.

cpp
Copy code
```
 seekg 0                tellg  sizeof
```

- These lines move the file pointer to the end of the file using `seekg` to get the file size in bytes. Then, it divides the file size by the size of a `stud` object to calculate the number of records in the file (`n`).

cpp
Copy code
```
if      0    write  char        sizeof              close   return
```

- If the file is empty (no records present), the function writes the contents of the `rec1` object (passed as an argument) to the file and returns after closing the file.

cpp
Copy code
```
  -1  while    0
```

- This loop iterates over the existing records in the file, starting from the last record.

cpp
Copy code
```
 seekg  sizeof               read  char       sizeof
```

- Inside the loop, it positions the file pointer at the `i`th record from the beginning of the file (`ios::beg`) and reads the `stud` object `rec` from the file.

cpp
Copy code
```
if                  seekp  1 sizeof               write  char
       sizeof                  else  break
```

- If the roll number of the current record (`rec.roll`) is greater than the roll number of the record to be inserted (`rec1.roll`), it moves the file pointer to the next position (`i+1`) and writes the current record `rec` back to the file. This operation shifts the existing records to make space for the new record to be inserted in ascending order based on roll numbers.

cpp
Copy code
```

```

- The loop continues moving towards the beginning of the file until it finds the correct position for insertion.

cpp
Copy code
```
      seekp  sizeof               write  char        sizeof              close
```

- After finding the correct position for insertion, it moves the file pointer to the `i`th position from the beginning of the file and writes the new record (`rec1`) to the file. Finally, it closes the file.

`student::display()` function step by step:

cpp
Copy code
```
void student::display          int  1              open  "stud.dat"
```

- This function is responsible for displaying the contents of the "stud.dat" file, which contains student records. It declares a local `stud` object `rec`, integers `i` and `n`, and a file stream object `f`. The file is opened in both input and output modes with binary mode to read binary data.

cpp
Copy code
```
   seekg  0              tellg   sizeof
```

- It moves the file pointer to the end of the file using `seekg()` and then determines the number of records in the file by dividing the current position of the file pointer (file size) by the size of a single `stud` object.

cpp
Copy code
```
   seekg  0
```

- This line resets the file pointer to the beginning of the file to start reading from the beginning.

cpp
Copy code
```
for   1                read  char         sizeof
          "\n"         "\t"         "\t"        "\t"
```

- It iterates through each record in the file. Inside the loop, it reads the `stud` object `rec` from the file using `read()`, and then it displays the attributes of the `rec` object (roll number, name, division, and address) using `cout`.

cpp
Copy code
```
   close
```

- Finally, it closes the file stream object `f` after displaying all the records.

This function essentially reads each student record from the file and prints its attributes to the console, effectively displaying the contents of the "stud.dat" file.

Let's break down the `student::search()` function:

cpp

Copy code

```
int student::search    int    0    open  "stud.dat"
    seekg  0    "\n\tEnter a Roll No: "
```

- This function is responsible for searching for a student record by roll number in the "stud.dat" file. It declares local variables `r` (for the roll number), `i` (as an index), and an input file stream object `fin`. It opens the file "stud.dat" in input mode with binary mode to read binary data and sets the file pointer to the beginning of the file.
- It prompts the user to enter a roll number to search for.

cpp
Copy code

```
while    read  char    sizeof    if    "\n\tRecord Found...\n"
    "\n\tRoll\tName\tDiv\tAddress"    "\n\t"    "\t"    "\t"
    "\t"    return  1
```

- Inside the loop, it reads each student record from the file using `read()` and compares the roll number (`rec.roll`) with the roll number entered by the user (`r`). If a match is found, it displays the record's details (roll number, name, division, and address) and returns the index `i+1`. The `i` variable is used to keep track of the position of the record being read from the file.

cpp
Copy code

```
    close    return  0
```

- After the loop, it closes the input file stream object `fin` and returns 0 if the record is not found.

Overall, this function allows the user to search for a student record by roll number in the file "stud.dat". If the record is found, it displays its details; otherwise, it returns 0 to indicate that the record was not found.

Let's break down the `student::Delete()` function:

cpp
Copy code

```
void student::Delete    int    "\nEnter Roll No to delete: "
```

- This function prompts the user to enter the roll number of the student record they want to delete.

cpp
Copy code

```
    open  "stud.dat"    open  "temp.dat"
```

- It declares input and output file stream objects `fin` and `fout`. `fin` is opened in input mode with binary mode to read binary data from the file "stud.dat". **`fout`** is opened in output mode with binary mode to write binary data to the temporary file "temp.dat".

cpp
Copy code
```
while   read  char   sizeof   if   write  char
sizeof
```

- Inside the loop, it reads each student record from the file "stud.dat" using **`read()`** and checks if the roll number (**`rec.roll`**) of the current record matches the roll number entered by the user (**`roll`**). If they do not match, it writes the record to the temporary file "temp.dat" using **`write()`**.

cpp
Copy code
```
  close   close   remove "stud.dat"
rename "temp.dat" "stud.dat"   "\n Record Deleted Successfully "
```

- After the loop, it closes both file stream objects **`fin`** and **`fout`**. Then, it removes the original file "stud.dat" using **`remove()`** and renames the temporary file "temp.dat" to "stud.dat" using **`rename()`**. This effectively replaces the original file with the temporary file, effectively deleting the record corresponding to the specified roll number.

Overall, this function allows the user to delete a student record by specifying its roll number. It achieves this by copying all records except the one to be deleted to a temporary file and then replacing the original file with the temporary file.

F24(sequential file)

1. **Struct Definition (`stud`)**: Defines a structure to hold student records, including roll number, name, division, and address.
2. **Class `student`**: Contains methods to insert a record (`insert`), display all records (`display`), search for a record (`search`), and delete a record (`Delete`). It also has a method `getdata` to get input from the user and return a `stud` structure.
3. **`getdata` Method**: Takes input for a student record and returns it.
4. **`insert` Method**: Inserts a record into the file `stud.dat` in a sorted order based on roll number. It reads existing records from the file, finds the correct position for the new record, and shifts existing records to accommodate the new record.
5. **`display` Method**: Displays all records in the file `stud.dat`.
6. **`search` Method**: Searches for a record based on the roll number entered by the user. It reads records from the file sequentially and compares the roll number until a match is found or the end of the file is reached.
7. **`Delete` Method**: Deletes a record from the file based on the roll number entered by the user. It reads records from the file, skips the record to be deleted, and writes the rest of the records to a temporary file. Finally, it replaces the original file with the temporary file.
8. **`main` Function**: Contains the main program loop to interact with the user. It provides options to create, display, delete, search for records, and exit the program. It uses a `do-while` loop to repeat the menu until the user chooses to exit.
9. **`seekg` (Seek Get)**:
    1. Syntax: `stream.seekg(offset, direction);`
    2. Sets the position in the file from which the next input operation will start.
    3. `offset` specifies the number of bytes from the specified `direction`.
    4. `direction` can be `ios::beg` (beginning of the file), `ios::cur` (current position in the file), or `ios::end` (end of the file).
10. **`seekp` (Seek Put)**:
    1. Syntax: `stream.seekp(offset, direction);`
    2. Sets the position in the file where the next output operation will start.
    3. `offset` specifies the number of bytes from the specified `direction`.
    4. `direction` can be `ios::beg` (beginning of the file), `ios::cur` (current position in the file), or `ios::end` (end of the file).

A1

Consider the telephone book database of N clients.

1. **Linear Probing**:
   - `linearprobing` **class**:
     - It has two private arrays, `phone` to store phone numbers and `flag` to indicate if a slot is occupied.
     - Constructor initializes all flags to 0.
     - `insert` **method**:
       - Inserts a phone number into the hash table using linear probing to find an empty slot.
       - If the slot is occupied, it probes the next slot until an empty slot is found.
     - `create` **method**:
       - Takes user input for the number of phone numbers to be added and adds them to the hash table using the `insert` method.
     - `find` **method**:
       - Searches for a phone number in the hash table using linear probing.
       - Returns the index if the phone number is found, else returns -1.
     - `search` **method**:
       - Takes user input for a phone number and calls the `find` method to search for it.
       - Prints whether the phone number is present in the hash table or not.
     - `print` **method**:
       - Prints the hash table showing the phone numbers stored in each slot or showing "-------" if a slot is empty.
2. **Separate Chaining**:
   - `separate_chaining` **class**:
     - It has an array of `node*` pointers to implement separate chaining.
     - Constructor initializes all pointers to `NULL`.
     - `create` **method**:
       - Takes user input for the number of elements to be added.
       - Adds each element to the hash table using separate chaining.
     - `insert` **method**:

        - Inserts a phone number into the hash table using separate chaining by appending a new node to the linked list at the hash index.
    - `display` **method**:
        - Displays the hash table, showing the linked lists at each index.
3. **Main Function**:
    - Creates instances of both classes and demonstrates their functionality by inserting phone numbers, displaying the hash tables, and searching for phone numbers.

D19 AVL

1. **AVLnode Structure**:
    - The `AVLnode` structure represents a node in the AVL tree.
    - It contains:
        - `cWord` and `cMean`: Strings representing the word and its meaning.
        - `left` and `right`: Pointers to the left and right child nodes.
        - `iHt`: An integer representing the height of the node in the tree.
2. **AVLtree Class**:
    - The `AVLtree` class represents the AVL tree itself.
    - It contains:
        - `Root`: A pointer to the root node of the AVL tree.
    - It provides various methods for manipulating the AVL tree, including:
        - `insert`: Inserts a new node into the AVL tree while maintaining its balance.
        - `deletE`: Deletes a node from the AVL tree while ensuring that the tree remains balanced.
        - Rotation methods (`LL`, `RR`, `LR`, `RL`): Perform rotations on the tree to balance it after insertions and deletions.
        - Utility methods (`height`, `bFactor`): Calculate the height of a node and its balance factor.
3. **Insertion (`insert` method)**:
    - The `insert` method recursively inserts a new node into the AVL tree.
    - If the tree is empty, it creates a new node and sets it as the root.

- If the word already exists in the tree, it prints a message indicating redundancy.
- After insertion, it updates the height of the current node and checks if the tree needs to be rebalanced using rotation methods (`LL`, `RR`, `LR`, `RL`).

4. **Deletion (`deletE` method)**:
    - The `deletE` method recursively deletes a node from the AVL tree.
    - If the node to be deleted has only one child or no child, it simply removes the node.
    - If the node to be deleted has two children, it finds the inorder successor (the smallest node in the right subtree), copies its data to the node to be deleted, and then deletes the inorder successor node.
    - After deletion, it updates the height of the current node and checks if the tree needs to be rebalanced using rotation methods.
5. **Balancing Factor (`bFactor` method)**:
    - The `bFactor` method calculates the balance factor of a node, which is the difference between the height of its left subtree and the height of its right subtree.
6. **Height (`height` method)**:
    - The `height` method calculates the height of a node in the AVL tree.
    - If a node is `NULL`, its height is considered to be -1.
7. **Rotation Methods (`LL`, `RR`, `LR`, `RL`)**:
    - These methods perform rotations on the AVL tree to balance it.
    - `LL` and `RR` are single rotations, while `LR` and `RL` are double rotations.
    - These rotations are used based on the balance factor of the nodes to maintain the AVL tree property.
8. **Traversal Methods (`inOrder`, `preOrder`)**:
    - The `inOrder` and `preOrder` methods are used to traverse the AVL tree in in-order and pre-order respectively, printing the word and its meaning.
9. **Main Function**:
    - The `main` function provides a menu-driven interface for users to interact with the AVL tree.
    - It allows users to insert new words, delete existing words, and display the tree using in-order and pre-order traversals.